Back

EMBEDDED C AND C++ COMPILER EVALUATION METHODOLOGY

Class 443, Embedded Systems Conference, September 26-30, 1999

Chuck Tribolet and John Palmer

The Authors

Chuck Tribolet is a Senior Programmer at IBM Research Division's Almaden Research Center in San Jose, California. Chuck graduated from Stanford in 1973 with a BS in Industrial Engineering and an MS in Computer Engineering. He joined IBM at Los Gatos in April 1973 and holds five patents on various aspects of software. He is currently responsible for evaluating C and C++ compilers for an assortment of embedded microprocessors for use in disk drives.

e-mail: triblet@almaden.ibm.com
Voice: 408-927-2707
Fax: 408-927-4166

John Palmer is a Research Staff Member at the IBM Almaden Research Center. He works on disk drive performance, both at the level of the overall drive and at the level of the processors that run the disk drives. For variety, he also studies disk error patterns to improve error recovery behavior. John graduated from Clemson University in 1969 with a BS in Applied Mathematics.

E-mail: jpalmer@almaden.ibm.com
Voice: 408-927-1724
Fax: 408-927-4010

Abstract

Making a processor architecture change is a frightening prospect to anyone in the embedded systems world. Not only are there questions about code porting and new development tools, but there are likely to be large unknowns about the performance of the new architecture. Nevertheless, advances in architecture knowledge and decreasing chip costs make it attractive to consider the potential gain from a more modern architecture.

In evaluating new architectures for our embedded application, we organized our efforts in two directions: code compilation and performance modeling. In the natural way, the two are related: generally the larger the code size, the longer it will take to execute and the more you will have to pay for memory space.

The compiler and microprocessor vendors will all supply data showing that their product is best. Which one is right? Asking them about the relative merits of

their product versus the competition isn't productive because they all think they have the greatest compiler in the world, and can show you data to prove it. Unfortunately, they spend more time turning the knobs on their own compiler than they do on their competitor's, so they may be comparing a well-tuned Mustang to a poorly tuned Camaro. Furthermore, their benchmark is different from your application. These in-house benchmarks have made them all honestly believe they have the best processor and compiler. Obviously, they ALL can't all have them.

This paper presents a methodology for evaluating embedded C and C++ compilers and their associated microprocessors. This methodology was used in the selection of the components to be used on a disk drive. It uncovered a number of potential pitfalls early in the design cycle. Had these been uncovered later, it would have been difficult to correct them.

We used our compiler observations in selecting the microprocessor. It was obvious that in some cases a good compiler was enhancing the performance of the microprocessor. Additionally, we inferred information on the performance characteristics of the microprocessor from analysis of the generated code.

The other tool we created to aid us in evaluating choices and in correctly tuning the final choice was a simulator that would predict the behavior of processors of different architectures. As input to the simulator, we had traces from two different processors running the different parts of our function that we hoped to combine on a single processor in a new architecture. With this simulator, we were able to predict with sufficient precision the behavior of our application on several different architectures, use this to predict the clock rates and memory systems that would be required with each of the processor architectures. We did it all long before we could have created running code for any of the proposed systems.

The paper describes the structure of the simulator itself, particularly how it was made flexible enough to model widely differing architectures. We will discuss the collection of traces and adaptation of them both to our expected future workload and to each of the proposed architectures. Finally, we will discuss the type of information obtained from the simulator and the way in which it was used in making an architecture decision, fine tuning the system that we picked, and in the planning for future evolution.

Compiler Selection

The Initial Approach

A naive approach is to select some code, compile for various engines or have the vendors compile it, and tabulate the results. We started out this way, but the actual process played out to be much more interesting.

Defining the Benchmark

The best benchmark is your entire application. Size does matter. The law of large numbers applies. Small benchmarks can produce misleading results if the modules selected happen to match what one combination of compiler and microprocessor handles well or poorly. Any vendor worth her salt can make a small benchmark look good. Furthermore, a small benchmark can be compiled on a smaller memory model than the entire application, will fit in a smaller Small Data Area than the whole, and needs fewer registers. Early in our evaluation, we created a single module benchmark. Its results were entirely different from those of a much larger benchmark.

Using your entire application as a benchmark will require some degree of modification, both to make it more ANSI compliant in the cases where it gratuitously diverges from the standard, and to isolate with preprocessor logic the noncompliant code in the cases that exploit some required nonstandard feature. A couple of person-days effort made 75% of our files compile. Three quarters of your application is a much better benchmark than any small benchmark or any vendor benchmark. Another couple of person-weeks over the course of the bench marking got the remaining 25% to compile. In our case, the cost of using the entire application as the benchmark paid off. It resulted in more accurate results, we now have a very nearly ANSI-compliant code base for our product, and it wasn't nearly as much work as predicted.

It is important to use your own application, or something very similar. Our application consists primarily of long strings of if/then/else statements so loop optimization is not as important as the optimization of linear code and branches, exploitation of the Small Data Area, optimization of register usage, and ROM size.

On the other hand a laser printer manufacturer would care about how fast the code will loop through a four hundred megabit image of a page at 600 dpi. Your benchmark must match the characteristics of your own application.

Since we did not have hardware with any of the alternative microprocessors, we chose not to actually exectute the code generated by the various compilers. If you are evaluating compilers for existing hardware, this would be a useful validation of the results.

Procuring Compilers

An old recipe for rabbit stew starts off: "First, catch a rabbit." Well, first identify and acquire the interesting compilers. Ask your microprocessor vendors which compiler vendor they recommend, but don't limit yourself to just one compiler per microprocessor. Review the trade press, and cruise the floor at the Embedded Systems Conference. Most compiler vendors will give you a 2-8 week evaluation license for free and they were very cooperative about extending these licenses when the selection process went longer than expected.

You will find that some compilers are "badge-engineered," that is, the same compiler will appear under several names. In general, you will only need to benchmark one of them in depth.

Measurement Environment

Once you have your first compiler in hand, it’s time to start setting up an efficient workbench for compiler measurement. You will be doing many, many, runs, so it is very important that it take as little of your time as possible to do a run. We ended up bench marking 22 compilers on 13 processors. One processor had six different compilers. We ran over 150 combinations of options.

There are variations in C between compilers and microprocessors. For example the size of an int might be 16 bits in one implementation and 32 bits in another, or one microprocessor might have a small data area sufficient to hold all static variables, while another’s SDA might be small and only hold a few key variables. I set up a single master source directory. When a given compiler required minor source code changes, I did it using preprocessor logic (#if statements) in that master source directory. These allowed minor changes (usually in pragmas and typedefs) to be tailored for each combination of compiler and microprocessor.

For each run on a given compiler there is a OBJ directory that contains the output files (object and listing files) and the variable portions of the input (the batch file used to invoke the compiler and parameter file, if any). I keep these object directories long term, which allows me to track what the environment was, and what the results were. Going a bit overboard on record keeping will pay dividends down the road (but bear in mind that I do sell disk drives for a living).

Running the Benchmark

Be aggressive about trying compiler options. The compilers each have many options, and it is important to arrive at the best set of the options for each compiler. To arrive at a starting set, examine the set of options, and determine which ones seem as if they might influence the resulting code. Then ask the compiler vendor what they suggest, select a starting set of options, and start running tests. Here’s where an efficient workbench and careful record keeping become critical. If the option helps, keep it. If it does not, discard it **for now**. Repeat this process until you have tried all the interesting options. Now go do it AGAIN because it’s possible that while option A didn’t help on the first pass, it will now that option B has been turned on. On the second pass, try turning off options you turned on during the first pass — it’s possible that some other change has now made option C a poor choice. Continue until you can’t make it better.

Take the vendor-suggested options with a large grain of salt. Almost every vendor has given me bad advice at one time or another. In particular, they all seem to believe that their compiler option to minimize code size was the best. In every case, I was able to beat this option by tweaking the individual compiler parameters.

A thorough tweaking of compiler options will frequently generate an improvement on the order of 30% over an initial decent set of options. If the initial set is truly abysmal, the improvement could be in excess of 100%. I remember one Pascal compiler from a prior life that you could degrade by 10,000% with the wrong options.

Pay attention to the memory model used and what is placed in the small data area. A large application won't fit in a tiny memory model or a tiny small data area, and you may not find out until you try linking the code. You may get a rude surprise (I did, in one case) at link time.

Comparing Compilers

As I have stated previously, the performance of our code is primarily influenced by the performance of linear code, and linear performance is largely determined by code size. Code size is also important due to ROM size constraints. At the beginning, the code was written a dialect of C that was specific to a particular compiler for a particular microprocessor (specifically Microsoft C6 for AMD 186). After a small amount of work, each of the new compilers would compile one half to two thirds of the modules, but different compilers failed on different modules. I developed a scheme where we would compare the size of the modules that DID compile to the size of the corresponding modules for the 186. For example, let us assume that we have three compilers, A (the current base), B, and C, and that we also have five modules (V, W, X, Y, and Z). If we compile each module with each compiler, and extract the module sizes, we get:

	Compiler A	Compiler B	Compiler C
Module V size	50	60	syntax error
Module W size	300	syntax error	250
Module X size	100	110	100
Module Y size	200	210	180
Module Z size	70	90	90
Compiled size		470	620
Corresponding compiler A size		420	670
Ratio (relative code size)		1.12	0.93

In this example, had we simply compared the sizes of what did compile (470 bytes versus 620 bytes), compiler C would have been unfairly penalized for successfully compiling the large module W, and we would have selected compiler B. However, when we compared the relative sizes of the successfully compiled modules, Compiler C is the winner.

Why not just compare the modules that compiled on both B and C? That would have had a smaller sample size (three modules) than the technique shown above (four modules).

Compiler Influences on Overall Performance

Some of the effects of the compiler on overall performance include:

1. Code size. This can be measured directly, usually by measuring the size of the .text section. Most compilers include a tool that will do this by reading the object file. Code size has a secondary effect of influencing the cache performance.

2. Instruction count can be measured directly, usually with a simple program that reads assembler listing files, or by dividing the code size by the average instruction size.

3. Cache behavior is more difficult to evaluate analytically. You can either run actual code on a target, or emulate.

4. SDA use is also more difficult to evaluate analytically in a direct manner, however its effects on performance are measurable in other ways. Poor SDA usage will results in additional memory accesses (which can be measured or simulated) poor locality of reference (whose influence on cache behavior can also be measured or simulated) and larger code size (which can be measured).

5. Register use is again more difficult to evaluate analytically, however its effects on performance are the similar to SDA use and can be measured in the same ways. You can get a feel for this by reading the generated assembler code.

6. Linking conventions affect performance by requiring more or fewer instructions to perform a function call, which influence code size, and by requiring more space in the stack, which can be determined by inspection of the generated code.

How does one assess performance? A basic view is that the amount of time (T_{total}) required to execute a program is the average time required to execute one instruction (T_{inst}) times the number of instructions required to do the job on that microprocessor:

$$T_{total} = T_{inst}*N_{inst}$$

How does one find the right values for T_{inst} and N_{inst}? A compiler study like the one we have been discussing does most of the work for you. Assuming that you can run your current code either on real hardware or a simulator and take traces of it, count the instructions in that trace. Next, determine the average instruction size for your benchmark. Do not accept the vendor's average instruction size. The product of these two numbers is the number of bytes of code executed. Now, you can use the relative code size across different compilers and microprocessors to generate a new "bytes of code" number that can be divided by the number of bytes per instruction for that environment to give N_{inst}.

A good value for T_{inst} is harder to generate. The time for an instruction is the cycle time of the machine times the number of cycles required by that instruction (cycles per instruction, CPI). If your environment is very simple (no caches or other unpredictable causes of processor stalls) then you can get CPI numbers from the architecture book. If you have a complex memory system, you will probably need a simulator to determine CPI.

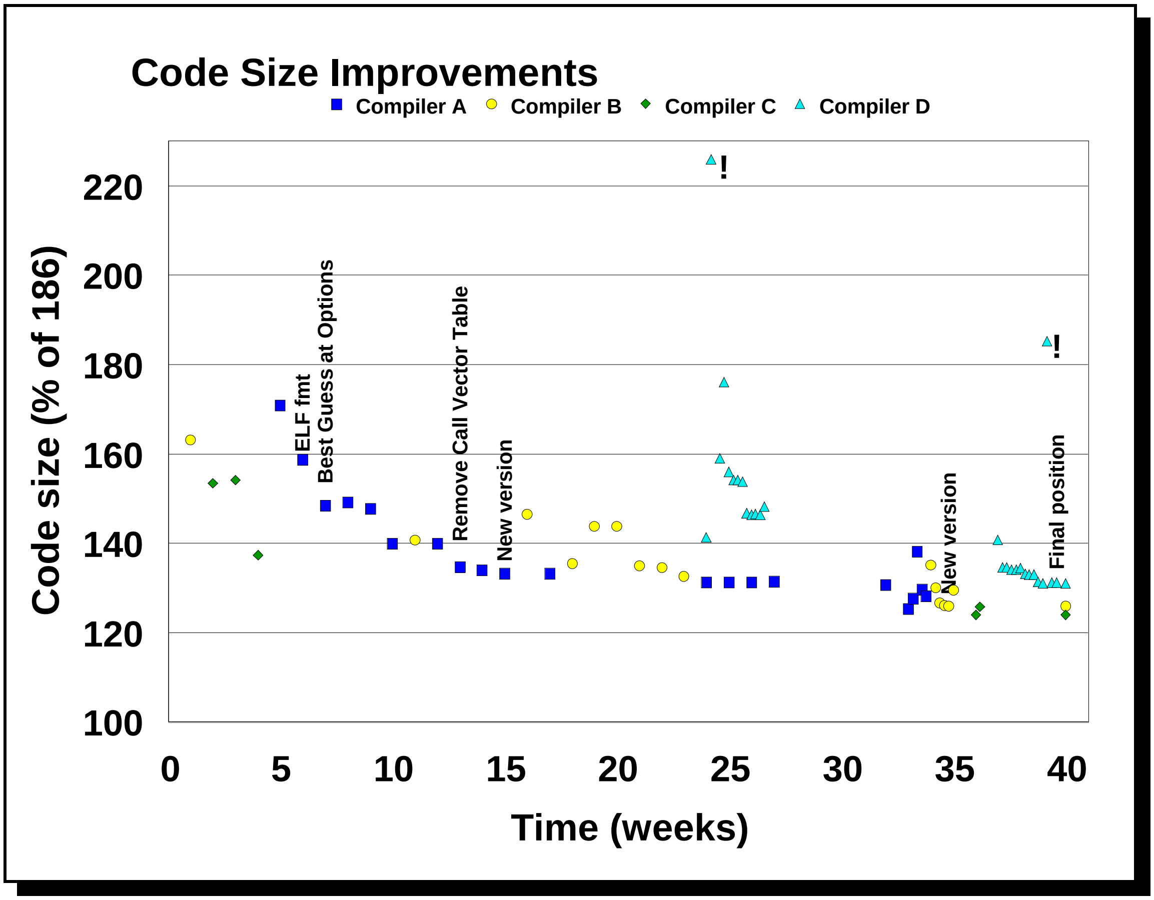
Compilers Can Get Better

“Greed is good,” and the compiler vendors are greedy. They want ALL the business, and they recognize that to win they have to be the best, or at least very close. Several of them, at various times, worked with us on joint studies of the generated code. In one case, this study resulted in a decrease in the size of the generated code on the order of 20%. Other studies produced smaller, but still significant, gains. A significant lever here is being able to tell vendor A that their code was behind the competition by X%. Warning: some evaluation licenses have nondisclosure clauses to prevent you from telling another vendor anything about the compiler being evaluated.

Greed also made the vendors listen and respond in weeks, not years. The embedded compiler vendors have a relatively small number of customers and this seems to have made them sensitive to satisfying the needs of each. Some were much better about this than others.

The comparison of multiple compilers on a single processor proved to be a valuable tool in pushing compiler vendors to improve their product. It’s difficult to generalize about processors that do not have this competition among compiler vendors. We saw one case where the only available compiler appeared to do very well, and another where we think the compiler was a problem.

If you are fighting to stay in a certain standard ROM size, working with your compiler vendor can be a very useful tool.



Results

The chart titled “Code Size Improvements” shows data points for several compilers, all on the same microprocessor. The vertical axis is code size relative to the existing 186 microprocessor. The horizontal axis is calendar time, approximately in weeks. Some significant things to note:

- All of the compilers improved greatly.
- Some of the improvements were from changing options.
- Some of the improvements were from improved versions of the compilers.
- Some of the option experiments were a BIG mistake. Note compiler D at about week 24.
- Except for one, they all ended up very close, and that vendor claims they have the problem solved, but I haven’t seen the code yet. Having compilers cluster in this fashion suggests that they may be reaching the optimal result for that processor architecture.
- Results on other microprocessors yielded similar curves. The processors themselves appeared to have more influence on code size than the choice of well-tuned compilers within a single processor family.

Performance Simulation

Overview

Our challenge was to provide performance predictions for half a dozen different processor architectures. Because of the rapidly changing costs of various types of on-chip memory, we needed to be able to assess a number of different memory structures attached to each processor. Finally, because of the cost of conversion, we needed to have at least rough plans for the new architecture to be extendable through several generations of drives, requiring predictions of code size and MIPS requirements through several years.

Our embedded application is disk drives and disk drives run two relatively independent types of applications. The servo application (moving the disk heads to the right place and keeping them there) is smaller in code size, but larger in number of instructions executed. This application is characterized by a high interrupt rate. Basically the servo application is an interrupt handler that gets a position error signal 5 to 10 thousand times per second and responds by changing the power applied to the disk arm. It is critically important that this interrupt handler run without delay because errors in execution timing can translate to positioning errors on the disk or even total confusion about head positioning. The other application is the command application which interprets read and write commands to the disk, instructs the servo to move the arm, and sets up all the other control hardware required to make a data transfer. The command application is generally much larger (in code size) than the servo application but the vast majority of this code is rarely used (error recovery, format, mode settings). The code which is run routinely as part of the basic disk read and write operations is generally relatively tolerant of changes in timing because most of its execution time is masked by the mechanical motions required of the arm.

In our current environment, the two applications run on separate processors. The servo code runs on a DSP and the command code runs in an Intel 186 architecture processor. It was our intention to provide a single engine that would be capable of running both of these applications and would be less expensive than our current multiprocessor design. We intended to do this by putting little-used code in inexpensive off-chip memory and locating high-usage code in a combination of on-chip memory and caches. We expected to be increasing the use of on-chip memory over the next few years, so we wanted to plan an architecture that would be adaptable to this faster memory as it became available.

A quick query to vendors showed that basically all of them had a clock rate range that would be likely to meet our needs. Each could also talk about several possibilities for cache sizes, speeds of multiply instructions, and various memory options. We needed a way to decide just how much money to invest in each of these in order to end up with a satisfactory system.

We briefly considered the possibility of using prototype boards, but rejected that approach because we didn't have the manpower to get enough code running in that environment to be meaningful. That approach also limits you to the memory structure that exists on the prototype board and would not allow evaluation of on-chip features that were critical to our designs. We also rejected using simulators provided by the vendors. While quite useful for certain questions, all of these were of a type that did a very precise simulation of the behavior of the processor and its pipeline, but like the prototype board, required code that would actually run correctly on that processor.

Realizing that we did not have the means to exploit the precise processor models, we chose an approach that provided minimum precision in the processor core, but did a relatively thorough job of modeling the memory system that feeds the processor. While we couldn't provide a sequence of actual instructions, we did think we could construct a plausibly representative sequence of instruction and data **addresses** that would suffice to exercise the memory system. With an address trace, you can track accesses to all of the various regions of memory, track cache contents, and make at least an approximation of bus contention. All of the architectures we were considering were more akin to RISC then CISC and they tended to execute most instructions in one cycle baring delays in instruction fetch or data access. Consequently, we could make a basic assumption that each instruction would take 1 cycle and we would attempt to track a couple of exception case multi-cycle instructions. The construction and use of this "address-based simulator" is the subject of the remainder of this section.

Trace Collection

For address-based simulation, you need addresses and these we generated from traces of our existing microcode. We collected traces for both our command application and our servo application. Both were simple in concept, but had "take care" aspects that bear mentioning. What one would like to do are just hook up a logic analyzer and have it trace all instruction fetches and all data references. In the case of our command application, we were able to follow that plan exactly because our processor is uncached and the memory bus comes out of the processor chip. We traced a collection of the common flavors of disk read and write operations and concatenated these traces to make a single trace. (We later did some work on infrequently used code like the format operation.) The only adaptation to the trace we made at this stage was to remove trace entries for idle time (we have an idle loop). This is important because including these instructions will tell you that the cache works very well when you have nothing to do, but that's not when you care, so simulate things you care about.

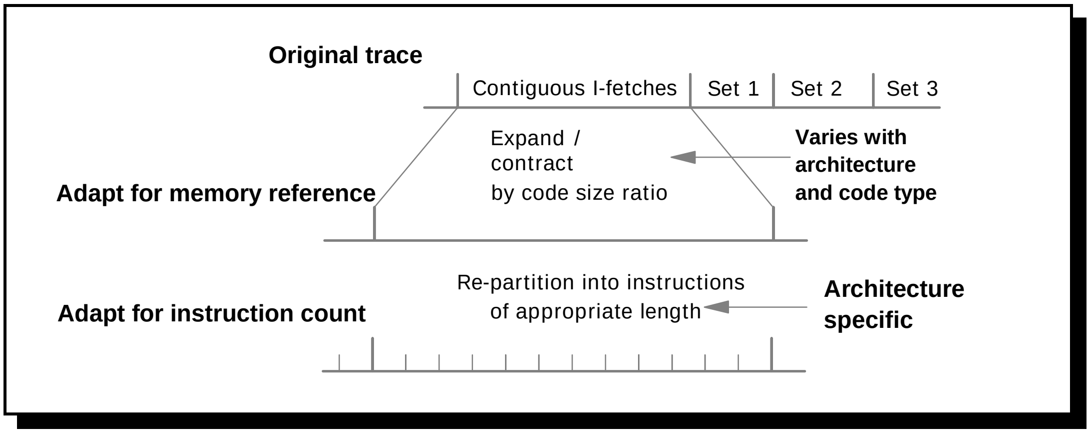
Our servo application was not so easy to trace because it was already built with on-chip memory and the processor memory bus was not visible from off-chip. We were fortunate that the servo programmers had built a simulated environment that they could run outside of the disk drive to do early debugging. We were able to use existing instrumentation in that simulation environment to extract an instruction trace and could infer a data trace from it. After doing a trace like this, a key step is to find a way to compare your trace to actual operation. In our case, we were able to compare the simulator's total time with real world total time for processing an interrupt. We discovered that the simulator had stubbed out about 20% of the total path length, so we would need to add that back in. (More about this later.)

We did not merge the two traces because we needed to preserve the separation so as to understand the interleaving of function under varying system conditions.

Trace Adaptation

We now had two traces, one coming from an Intel 186 architecture and one coming from a TI DSP. We needed to convert them into traces that could have come from each of the new architectures. We also needed to extend the traces to account for future function additions or for those functions that were for various reasons missing from the traces.

The architectural conversion of a trace is done independently on instruction and data references. To convert the instruction references to a new architecture, we followed a 4-step process. First, scan the trace to locate areas of contiguous I-fetches. These would correspond to the "basic blocks" within the microcode (i.e., sections of code between jumps or branches). Second, expand the size of the block by an architecture-related code size ratio. The code size ratio's for each architecture were determined by the compiler study already described. Third, assign a new starting address to the block, also based on the code size ratio. This maintains the same relative positioning for the new and old versions of the block. Fourth, divide up the newly re-sized block of code into instructions of length(s) appropriate to the target architecture and insert each of these instruction addresses in the new trace. During this process, data references are carried through without changing the data address, leaving the references as much as possible in the same position relative to the instruction sequence.

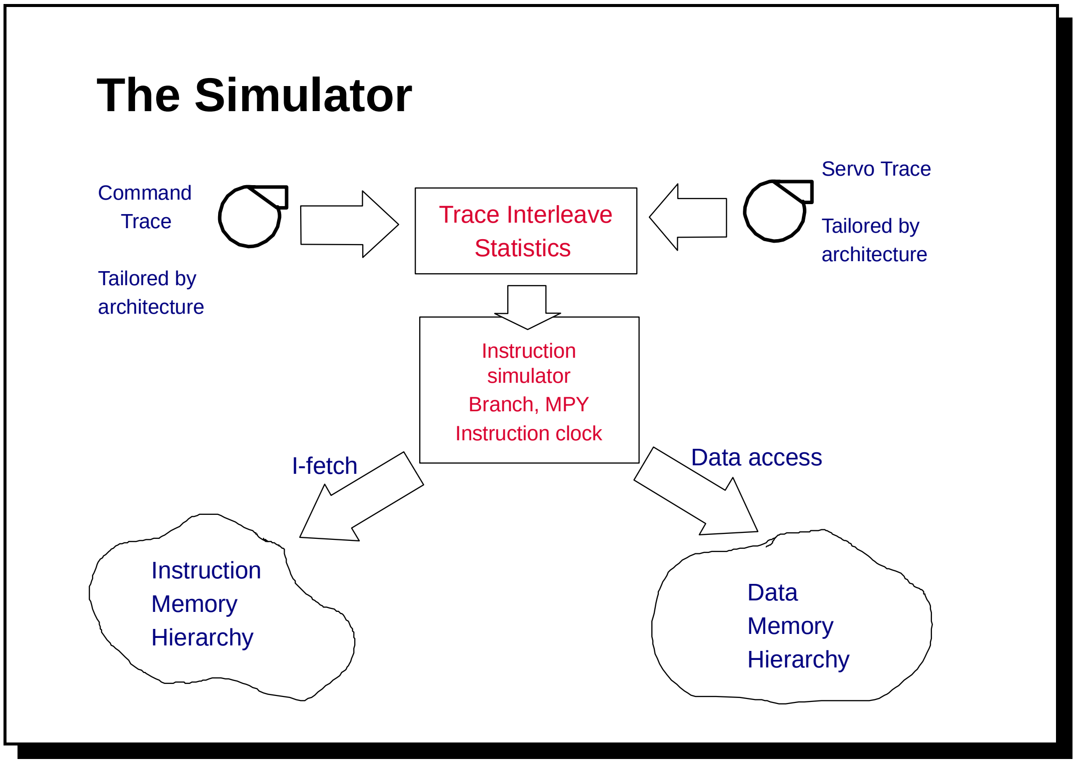

Original trace
Contiguous I-fetches
Set 1
Set 2
Set 3
Expand / contract by code size ratio
Varies with architecture and code type
Adapt for memory reference
Re-partition into instructions of appropriate length
Architecture specific
Adapt for instruction count

Data reference conversion is a little more complex because it requires adaptation for the number of registers on the target architecture. Both of our source traces were taken from accumulator-based machines, so those traces exhibit a lot of memory loads and stores that would not appear if more registers are available. At first, we ignored the effect of the extra memory references, assuming that single-cycle caches and processor pipelines would make these disappear. As we learned more about our target machines, however, we discovered that several had extra cycle penalties for load or store operations or both. At that point, we went back and built a simple LRU model for moving data into and out of registers. In this model, data was put into a register when first referenced and would not be re-fetched from memory as long as it was in a that register. Data in a register would be discarded or written back to memory when it was the oldest value in a register and a register was required for new data. Such a model, if given access to all the registers in the machine, would give far too good a picture of register effectiveness. To calibrate the model, we went to the code generated in the compilation exercises. In a subset of the modules, we manually tabulated the density of memory references in the generated code. Armed with this “memory reference density”, we went back to the register model and decreased the number of registers available to it until it yielded a set of load and store operations equal to that in the compiled code. What we found was that “clairvoyant” use of 4-5 registers generated a density of data references equivalent to the compiled code, even on machines with many more registers.

The last piece of trace adaptation was artificial trace expansion to cover function additions or, in the case of the servo trace, existing functions that were not captured in the trace. To do this, we simply took the converted trace and replayed it for an appropriate distance, changing the base addresses of both code and data to a new region to simulate new code and variables.

Simulator Structure

With the traces in hand, it was time to have a simulator. As shown below, at the center of the simulator is the “Instruction simulator” that runs the master clock and handles the small number of exceptions to the one cycle-per-instruction assumption. The Instruction Simulator is fed trace records by an interleave routine that takes input alternately from the command trace and the servo trace. The Instruction Simulator then indicates instruction fetch by sending the address of the instruction out on the “I-fetch” path and data access by sending a read or write out the “Data access” path. Each of these memory paths consists of multiple elements that can be any combination of busses and caches leading ultimately to a memory of some type.



The Simulator

Command Trace

Tailored by architecture

Trace Interleave Statistics

Servo Trace

Tailored by architecture

Instruction simulator
Branch, MPY
Instruction clock

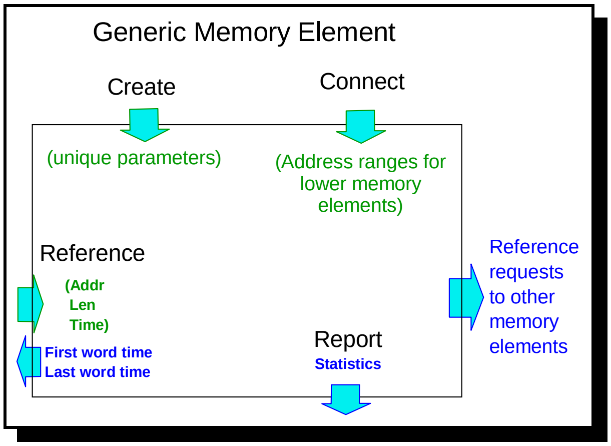
I-fetch

Data access

Instruction Memory Hierarchy

Data Memory Hierarchy

The precise definition of the instruction and data memory hierarchies is the real key to this simulator because it is in these places that we expected to see the largest performance effects and where we would also see the largest number of choices that needed evaluating. To allow the dynamic configuration of a large number of different architectural arrangements, we created each element of the memory system as a subclass of a generic memory class. The parent class had roughly the interface shown below in which lots of device-specific parameters are passed when the object is created. Following creation, elements receive a set of calls that identify the memory elements below them and address ranges supported by each. When actual simulation starts each element will receive only generic memory requests to read or write data. Each element of the hierarchy is designed to accept requests from above and pass them on to other elements below. An element may add time both before passing the request on and after receiving a response.



Generic Memory Element

Create

(unique parameters)

Connect

(Address ranges for lower memory elements)

Reference

(Addr Len Time)

First word time
Last word time

Report

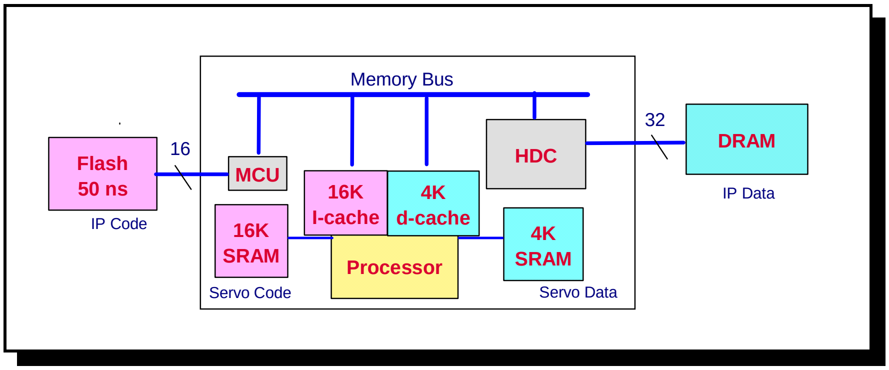
Statistics

Reference requests to other memory elements

Within this general structure, we built elements to simulate the timing and arbitration priorities of busses, cache elements that worked as instruction or data caches, and memory elements that would exhibit the behavior of flash, SRAM, and SDRAM. The Bus elements we built were more often coded uniquely rather than being specified by parameters. The cache element accepts parameters like total size, line size, associativity, and replacement algorithm. We built memory elements for each of the basic types of memory but then parameterized them with respect to address range, width, and various speed characteristics.

As an example, consider the operation of the cache element. At creation time, the initialization routine gets the total size, line size, associativity, and replacement algorithm. It uses the size and associativity arguments to allocate the arrays that will track the contents of each cache line. It then will read from a disk file the contents of the cache of that type and shape at the end of the previous simulation run (i.e., the last instruction cache of 16K bytes with line size 16 and associativity 2). We found it useful to save and restore the cache state rather than have the early part of the simulation distorted by startup effects. With everything in place, the first request arrives. If it is a hit in the cache, the response is made that the data is immediately available. If the data is not in the cache, then the cache will send a request for the appropriate cache line addresses to the next memory element (probably the memory bus). When that request is returned, it will have the time at which the first word in that line becomes available and when the last line is available. If the first word is the requested word (an architecture question), that word is returned without further delay, but the cache element remembers that the current line is still in the process of being loaded. If requests for other lines arrive immediately, they can be handled. If, as is often the case, the request is for another word in the line being fetched, then the cache must decide how much delay to add in its response.

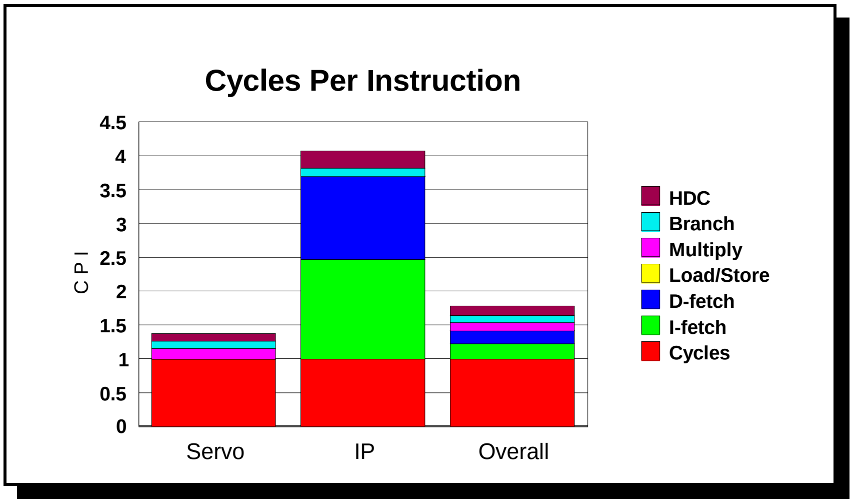
In this fashion, an instruction fetch request made by the Instruction Simulator will travel some distance into the instruction memory hierarchy before being returned and at the end it will return a time at which that instruction will be available. A typical instruction hierarchy might consist of (1) the processor prefetch queue, (2) an instruction cache, (3) the memory bus, and (4) a flash memory. The illustration below is a typical configuration for our application. In it, you can see that we have multiple paths for instruction fetch, one to on-chip memory and one to off-chip flash. We actually have four paths to data memory, one to on-chip SRAM, one to the flash, one to off-chip DRAM, and one to registers in the Hard



Disk Controller (HDC). ASIC registers we generally defined as an SRAM with appropriate timing.

Simulator Output

Running the simulator produced for us, as “management output”, an indicator of whether performance was sufficient. Of much greater import was further information about the performance of the various components of this specific configuration. Among these, it was particularly useful to have overview information about sources of delay and we typically did this with a breakout of CPI (Cycles Per Instruction). In such these bar charts, the bottom 1 is our basic one cycle per instruction assumption. The chart below shows that the servo code is executing with little delay, but the command code is running about three times slower because instruction fetch and data access delays. Clearly this code that is running from cache and obviously getting enough cache misses to cause a significant slow down. Without knowing more about the environment you can’t say whether it needs to be fixed and if so, whether the best way to improve things would be to increase the cache size or to change the memory backing the cache. In this particular case, actually, it was fast enough for our purposes so more cache space or memory improvements would have been an unnecessary expense. A couple more simulator runs told us how much cache would be required to make the large I-fetch and data access delays go away if we needed the extra speed.



This type of analysis was typical of our use of the simulator: postulate a configuration, run the simulator to define performance, identify opportunity areas if better performance is needed, and then try several alternatives to find the most attractive one.

In addition to the overview of delay sources, we always had the simulator generate output on the specific performance of each component. For a cache, that might look like the following:

```
I-cache results
      Total:    383034
      Hits:     368179
      Misses:    14855
      Hit Ratio:    96.1 %
      Hit times:   18953 usec.    Average:  51.5 nsec
      Miss times:   6355 usec.    Average: 427.8 nsec
```

What You Can Use It For

We used the simulator for LOTS of things. It helped us identify functional configurations for every vendor. Given a successful configuration, it then helped to identify a lowest cost successful configuration. It allowed us to play “what if” games with possible future memory technologies (or prices). It gave us a way to answer the processor vendors when they asked for advice on cache sizes and shapes. It gave us ammunition is a couple of cases to sway their architects to our way of thinking on the importance of certain memory hierarchy approaches. (We discovered that the vendors generally only get relatively confused direction from the users of their processors, so if you walk in with this kind of detailed understanding, they tended to listen.)

Most of the issues we addressed were specific to our application and environment, but there were a number of observations that seem to be broadly applicable:

- If you will be using a cache, making sure it works well enough should be your number 1 priority. Cache success comes from low miss rate, a fast line fetch time, or both. Fix performance with whichever is less expensive.
- Watch the clock speed issue. It can be misleading. Processors with smaller instruction sizes often need higher clock rates than those with larger instruction sizes. Code size often (but not always) goes down with smaller instructions, but seldom as much as the decrease in instruction size. Consequently, you end up needing more of the little instructions and hence a higher clock rate to do the job in the same time.
- Don’t assume your silicon vendor understands your requirements and potential uses for on-chip memory. Some were better than others in this area.
- The numbers you get from the vendor on cycles per instruction (or instructions per cycle) are achievable on some workload but almost certainly not on yours.

A warning to the unwary is appropriate. Simulators require large amounts of skepticism, particularly when you are predicting the future. Every time you change anything, even an input parameter, you need to assume that the simulator has gone haywire and is generating spurious results, particularly if those results look interesting. Any time you get an “interesting” result, you must have enough intermediate or component timing output so that you can identify the precise features that are causing the interesting behavior and, from that, decide whether or not you will accept the simulator as being realistic. In my experience, even the most unexpected results are easy to explain once you get them and spend a little time looking at component performance. If you don’t get a simple explanation, it is HIGHLY likely that the simulator is lying to you.

Even with the risks of being misled, a good simulator is often the only best chance for making the right first guess at what silicon to build. Particularly across major shifts like a change of architecture, it can provide insights that will save much valuable time in rebuilds avoided. So, good luck in your endeavors and may the simulator be with you.