The Direct Cost of Virtual Function Calls in C++

Karel Driesen and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
{karel,urs}@cs.ucsb.edu
http://www.cs.ucsb.edu/{~karel,~urs}

Abstract. We study the direct cost of virtual function calls in C++ programs, assuming the standard implementation using virtual function tables. We measure this overhead experimentally for a number of large benchmark programs, using a combination of executable inspection and processor simulation. Our results show that the C++ programs measured spend a median of 5.2% of their time and 3.7% of their instructions in dispatch code. For "all virtuals" versions of the programs, the median overhead rises to 13.7% (13% of the instructions). The "thunk" variant of the virtual function table implementation reduces the overhead by a median of 21% relative to the standard implementation. On future processors, these overheads are likely to increase moderately.

1. Introduction

Dynamic dispatch, i.e., the run-time selection of a target procedure given a message name and the receiver type, is a central feature of object-oriented languages. Compared to a subroutine call in a procedural language, a message dispatch incurs two kinds of overhead: a *direct cost* and an *indirect cost*.

The *direct cost* of dynamic dispatch consists of the time spent computing the target function as a function of the run-time receiver class and the message name (selector). The ideal dispatch technique would find the target in zero cycles, as if the message send was a direct procedure call. Thus, we define the *direct cost* of dynamic dispatch for a particular program P as the number of cycles spent on the execution of P, minus the number of cycles spent on the execution of an equivalent program P_{ideal} in which all dispatches are replaced by direct procedure calls that magically invoke the correct target function. In practice, P_{ideal} may be impossible to construct, since some sends will have varying targets at run time. However, a dispatch technique may reach ideal performance (zero overhead) on some programs on a superscalar processor, as we will discuss in section 2.6.

The *indirect cost* stems from optimizations that cannot be performed because the target of a call is unknown at compile time. Many standard optimizations such as interprocedural analysis require a static call graph to work well, and many intraprocedural optimizations are ineffective for the small function bodies present in object-oriented programs. Thus the presence of dynamic dispatch hinders optimization, and consequently, the resulting program will run more slowly. Although indirect costs can be an important part of the total overhead [HU94] this study will mostly ignore them and instead focus on the direct costs.

The aim of this study is to measure the direct cost of virtual function table lookup for a number of realistic C++ programs running on superscalar processors, and to identify the processor characteristics that most affect this cost. Unfortunately, it is hard to measure this cost directly since we cannot usually run program P_{ideal} (the program without any dispatch code). Although it is fairly easy to count the number of instructions executed on behalf of dynamic dispatch, this measure does not accurately reflect the cost in processor cycles. On modern pipelined processors with multiple instruction issue the cost of an

OOPSLA 96 - 10/96, San Jose, CA USA

instruction may vary greatly. For example, on a 4-way superscalar processor with a branch penalty of 6, an instruction can take anywhere between 0.25 and 7 cycles[†].

Therefore we measure the direct cost of virtual function table lookup by *simulating* the execution of P and P_{ideal}. Using an executable editor and a superscalar processor simulator, we compute the execution times of both programs, thus arriving at the direct cost of dispatch. In addition to allowing dispatch cost to be measured at all, simulation also facilitates exploring a broad range of possible processor implementations, thus making it possible to anticipate performance trends on future processors.

Our measurements show that on a processor resembling current superscalar designs, the C++ programs measured spend a median of 5.2% and a maximum of 29% of their time executing dispatch code. For version of the programs where every function was converted to a virtual function, the median overhead rose to 13.7% and the maximum to 47%.

The rest of this paper is organized as follows: section 2 provides some background on virtual function table lookup and the aspects of superscalar processors that are relevant in this context. Section 3 discusses our experimental approach and the benchmark programs, and section 4 presents the experiments, and section 5 discusses their results.

2. Background

This study concentrates on the dispatch performance of C++ programs on modern (superscalar) hardware. While we assume that the reader is familiar with the general characteristics of C++, we will briefly review its most common dispatch implementations, virtual function tables and the "thunk" variant, and the salient hardware characteristics of modern processors. Readers familiar with these topics may wish to skip to section 2.5.

[†] In the absence of cache misses.

2.1 Virtual function tables

C++ implements dynamic dispatch using virtual function tables (VFTs). VFTs were first used by Simula [DM73] and today are the preferred C++ dispatch mechanism [ES90]. The basic idea of VFTs is to determine the function address by indexing into a table of function pointers (the VFT). Each class has its own VFT, and each instance contains a pointer to the appropriate VFT. Function names (selectors) are represented by numbers. In the single-inheritance case, selectors are numbered consecutively, starting with the highest selector number used in the superclass. In other words, if a class C understands *m* different messages, the class' message selectors are numbered 0..*m*-1. Each class receives its own dispatch table (of size *m*), and all subclasses will use the same selector numbers for methods inherited from the superclass. The dispatch process consists of loading the receiver's dispatch table, loading the function address by indexing into the table with the selector number, and jumping to that function.

With multiple inheritance, keeping the selector code correct is more difficult. For the inheritance structure on the left side of Figure 1, functions c and e will both receive a selector number of 1 since they are the second function defined in their respective class. D multiply inherits from both B and C, creating a conflict for the binding of selector number 1. In C++ [ES90], the conflict is resolved by using multiple virtual tables per class. An object of class D has two dispatch tables, D and Dc (see Figure 1).[‡] Message sends will use dispatch table D if the receiver object is viewed as a B or a D and table Dc if the receiver is viewed as a C. The dispatch code will also adjust the receiver address before calling a method defined in C [ES90].

Figure 2 shows the five-instruction code sequence that a C++ compiler typically generates for a virtual function call. The first instruction loads the receiver object's VFT pointer into a register, and the subsequent two instructions index into the VFT to load the target address and the receiver pointer adjustment (delta) for

[‡] We ignore virtual base classes in this discussion. Our benchmark suite contains only a few instances of them, not enough to allow meaningful measurements. Virtual base classes introduce an extra overhead of a memory reference and a subtraction [ES90].

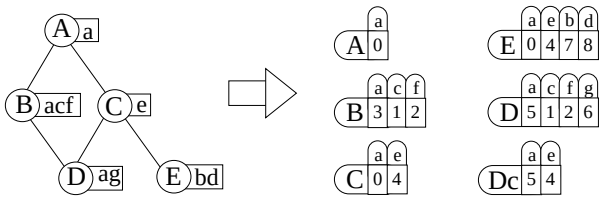


Figure 1. Virtual function tables (VFTs) Capital characters denote classes, lowercase characters message selectors, and numbers method addresses

```
1: load [object_reg + #VFToffset], table_reg
2: load [table_reg + #deltaOffset], delta_reg
3: load [table_reg + #selectorOffset], method_reg
4: add  object_reg, delta_reg, object_reg
5: call  method_reg
```

Figure 2. Instruction sequence for VFT dispatch

multiple inheritance. The fourth instruction adjusts the receiver address to allow accurate instance variable access in multiple inherited classes. Finally, the fifth instruction invokes the target function with an indirect function call.

2.1.1 Thunks

Instructions 2 and 4 in Figure 2 are only necessary when the class of the receiver has been constructed using multiple inheritance. Otherwise, the offset value loaded into the register *delta_reg* in instruction 2 is zero, and the add in instruction 4 has no effect. It would be convenient if we could avoid executing these useless operations, knowing that the receiver's class employs only single inheritance. Unfortunately, at compile time, the exact class of the receiver is unknown. However, the receiver's virtual function table, which stores the offset values, "knows" the exact class. The trick is to perform the receiver address adjustment only after the virtual function table entry is loaded. In the GNU GCC thunk implementation, the virtual function table entry contains the address of a parameterless procedure (a thunk), that adjusts the receiver address and then calls the correct target function (see Figure 3). In the single inheritance case, the virtual function table entry points directly to the

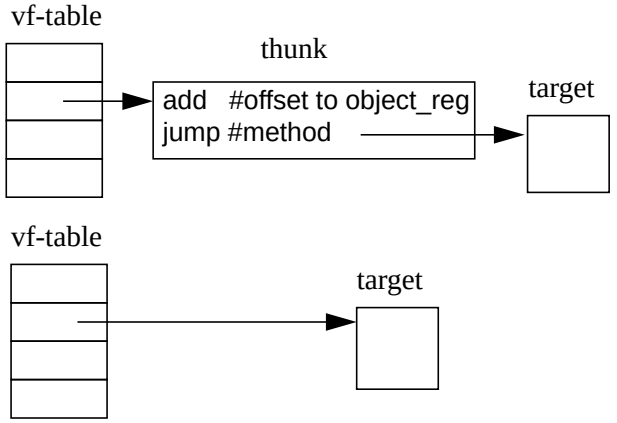


Figure 3. Thunk virtual function tables, in the multiple inheritance case (above), and the single inheritance case (below)

target function. Instead of always loading the offset value and adding it to the *this* pointer, the operation only happens when the offset is known to be non-zero. Since multiple inheritance occurs much less frequently than single inheritance, this strategy will save two instructions for most virtual function calls†. Therefore, barring instruction scheduling effects, thunks should be at least as efficient as standard virtual function tables.

2.2 Superscalar processors

How expensive is the virtual function call instruction sequence? A few years ago, the answer would have been simple: most instructions execute in one cycle (ignoring cache misses for the moment), and so the standard sequence would take 5 cycles. However, on current hardware the situation is quite different because processors try to exploit instruction-level parallelism with *superscalar execution*. Figure 4 shows a simplified view of a superscalar CPU. Instructions are fetched from the cache and placed in an instruction buffer. During every cycle, the issue unit selects one or more instructions and dispatches them to the appropriate functional unit (e.g., the integer unit).

The processor may contain multiple functional units of the same type. For example, the processor in Figure 4 has three integer units and thus can execute up to three

† In the GNU GCC implementation for SPARC excutables, one of the offset instructions is usually replaced by a register move. The latter is necessary to pass the *this* pointer in register %o0 to the callee.

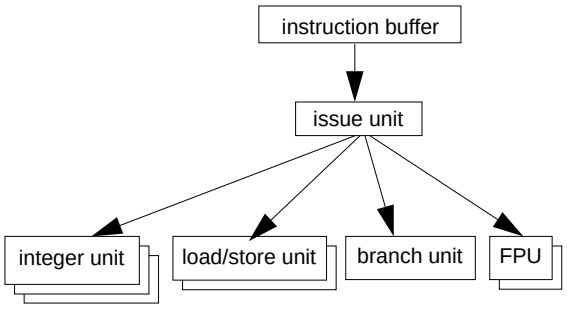


Figure 4. Simplified organization of a superscalar CPU

integer instructions concurrently. The number of instructions that can be dispatched in one cycle is called the issue width. If the processor in Figure 4 had an issue width of four (often called “four-way superscalar”), it could issue, for example, two integer instructions, one load, and a floating-point instruction in the same cycle.

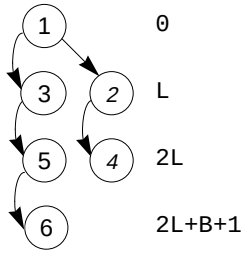
Of course, there is a catch: two instructions can only execute concurrently if they are independent. There are two kinds of dependencies: data dependencies and control dependencies. *Data dependencies* arise when the operands of an instruction are the results of previous instructions; in this case, the instruction cannot begin to execute before all of its inputs become available. For example, instructions 2 and 3 of the VFT dispatch sequence can execute concurrently since they are independent, but neither of them can execute concurrently with instruction 1 since they both use the VFT pointer loaded in instruction 1.

The second form of dependencies, *control dependencies*, result from the fact that some instructions influence the flow of control. For example, the instructions following a conditional branch are not known until the branch executes and determines the next instruction to execute (i.e., whether the branch is taken or not). Therefore, even if an instruction after the branch has no data dependencies, it cannot be executed concurrently with (or before) the branch itself.

Both forms of dependencies may carry an execution time penalty because of pipelining. Whereas the result of arithmetic instructions usually is available in the next cycle (for a latency of one cycle), the result of a load issued in cycle *i* is not available until cycle *i+2* or *i+3* (for a *load latency L* of 2 or 3 cycles) on most current processors even in the case of a first-level cache hit. Thus, instructions depending on the loaded value cannot begin execution until L cycles after the load. Similarly, processors impose a *branch penalty* of B cycles after conditional or indirect branches: when a branch executes in cycle *i* (so that the branch target address becomes known), it takes B cycles to refill the processor pipeline until the first instruction after the branch reaches the execute stage of the pipeline and produces a result.

To summarize, on ideal hardware (with infinite caches and an infinite issue width), the data and control dependencies between instructions impose a lower limit on execution time. If N instructions were all independent, they could execute in a single cycle, but if each of them depended on the previous one they would take at least N cycles to execute. Thus, *the number of instructions is an inaccurate predictor of execution time on superscalar processors*. Even though actual processors do not have infinite resources, this effect still is significant as we shall see later in this paper.

Figure 5 shows the dependencies between the instructions of the VFT dispatch sequence. At most two instructions can execute in parallel, and the minimum cost of the entire sequence is 2L for the chain of data dependencies between instructions 1, 3, and 5, and B for the branch penalty for instruction 5, i.e., the




```
1: load [object_reg + #VFToffset], table_reg
2: load [table_reg + #deltaOffset], delta_reg
3: load [table_reg + #selectorOffset], method_reg
4: add  object_reg, delta_reg, object_reg
5: call  method_reg
```

Figure 5. VFT execution schedules with cycle counts and assembly code.
Dependencies are indicated with arrows.

delay until the first instruction after it can execute. Thus, the sequence's execution time on a processor with load latency L and branch penalty B is 2L + B + 1 cycles.

In a previous study [DHV95] we approximated the dispatch cost of several techniques by analyzing the call sequence carefully and describing their cost as a function of load latency and branch penalty, taking into account superscalar instruction issue. However, this approximation (e.g., 2L + B + 1 for VFT dispatch) is only an upper bound on the true cost, and the actual cost might be lower. The next few sections explain why.

2.3 Branch prediction

Since branches are very frequent (typically, every fifth or sixth instruction is a branch [HP95]) and branch penalties can be quite high (ranging up to 15 cycles on the Intel Pentium Pro processor [Mic95]), superscalar processors try to reduce the average cost of a branch with branch prediction. Branch prediction hardware guesses the outcome of a branch based on previous executions and immediately starts fetching instructions from the predicted path. If the prediction is correct, the next instruction can execute immediately, reducing the branch latency to one cycle; if predicted incorrectly, the processor incurs the full branch penalty B. Predictions are based on previous outcomes of branches. Typically, the branch's address is used as an index into a prediction table. For conditional branches, the result is a single bit indicating whether the branch is predicted taken or not taken, and typical prediction hit ratios exceed 90% [HP95].

For indirect branches, the prediction mechanism must provide a full target address, not just a taken/not taken bit. A *branch target buffer* (BTB) accomplishes this by storing the predicted address in a cache indexed by the branch address (very similar to a data cache). When the processor fetches an indirect branch, it accesses the BTB using the branch instruction's address. If the branch is found, the BTB returns its last target address and the CPU starts fetching instructions from that address before the branch is even executed. If the prediction is wrong, or if the branch wasn't found, the processor stalls for B cycles and updates the BTB by storing the branch and its new target address.

BTBs affect the cost of the VFT dispatch sequence: if the virtual call was executed previously, is still cached in the BTB, and invokes the same function as in the previous execution, the branch penalty is avoided, reducing the sequence's cost to 2L + 1.

2.4 Advanced superscalar execution

Unfortunately, the truth is even more complicated. To improve performance, modern processors employ two additional techniques that can decrease the performance impact of dependencies.

First, instructions may be executed *out of order*: an instruction *I* that is waiting for its inputs to become available does not stall all instructions after it. Instead, those instructions may execute *before I* if their inputs are available. Additional hardware ensures that the program semantics are preserved; for example, if instructions I_1 and I_2 write the same register, I_1 will not overwrite the result of I_2 even if I_2 executes first. Out-of-order execution increases throughput by allowing other instructions to proceed while some instructions are stalled.

Second, *speculative execution* takes this idea one step further by allowing out-of-order execution across conditional or indirect branches. That is, the processor may speculatively execute instructions before it is known whether they actually should be executed. If speculation fails because a branch is mispredicted, the effects of the speculatively executed instructions have to be undone, again requiring extra hardware. Because branches are so frequent, speculating across them can significantly improve performance if branches can be predicted accurately.

Of course, the processor cannot look arbitrarily far ahead in the instruction stream to find instructions that are ready to execute. For one, the probability of fetching from the correct execution path decreases exponentially with each predicted branch. Also, the issue units must select the next group of instructions to be issued from the buffer within one cycle, thus limiting the size of that buffer. The most aggressive

designs available today select their instructions from a buffer of about 30-40 instructions [Mic94][Mic95], so that instructions have to be reasonably "near" the current execution point in order to be issued out-of-order.

2.5 Co-scheduling of application code

With speculative, out-of-order execution the cost of the VFT dispatch sequence is not only highly variable (depending on the success of branch prediction), but it cannot be computed in isolation from its surrounding code. For example, if many other instructions precede the dispatch sequence, they could execute during the cycles where the processor would otherwise lay idle waiting for the loads to complete. Or vice versa, the dispatch instructions could fit into empty issue slots of the rest of the basic block. This co-scheduling of the application and dispatch code may reduce the overall cost significantly, possibly to the point where completely removing the dispatch code would not speed up the program at all (since all dispatch instructions fit into otherwise empty issue slots). Thus, at least in theory, a dispatch implementation may reach zero overhead (i.e., adds no cycles to the execution time) even though it does introduce extra instructions.

2.6 Summary

While all of the processor features discussed above improve performance on average, they also increase the variability of an instruction's cost since it depends not only on the instruction itself (or the instruction and its inputs), but also on the surrounding code. Most processors sold today (e.g., the Intel Pentium and Pentium Pro processors, as well as virtually all RISC processors introduced since 1995) incorporate several or all of these features. As a result, it is hard to predict how expensive the average C++ virtual function call is on a current-generation PC or workstation. The experiments described in the rest of this paper aim to answer exactly this question.

3. Method

This section describes how we simulated the execution of the C++ programs, what processor features we assumed, and what benchmarks we used.

3.1 Simulation scheme

Figure 6 shows an overview of our experimental approach: first, the C++ program compiled by an optimizing compiler (we used GNU gcc 2.6.3 and 2.7.2 with options -O2 -msupersparc). Then, an application that uses the EEL executable editing library [LS95] detects the dispatch instructions and produces a file with their addresses. Using this file as well as a processor description, the superscalar processor simulator then runs the benchmark.

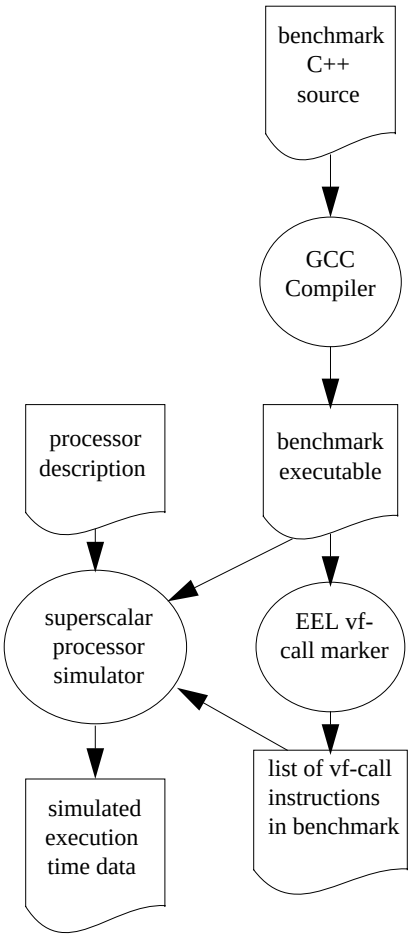


Figure 6. Overview of experimental setup

The simulator can execute most SPARC programs using the *shade* tracing tool [CK93]. Shade always executes all instructions of the program so that programs produce the same results as if they were executed on the native machine. Each instruction executed can be passed to a superscalar processor simulator that keeps track of the time that would be consumed by this instruction on the simulated processor. Optionally, the simulation of dispatch instructions can be suppressed (i.e., they are executed but not passed to the timing simulator), thus simulating the execution of P_{ideal} , the program using the perfect, zero-cost dynamic dispatch scheme.

Although we currently use only benchmarks for which we have the source, this is not strictly necessary. Provided that the vf-call marker program detects all virtual calls correctly, any executable can be measured. The source language does not even have to be C++, as long as the language under consideration uses VFT dispatch for its messages. Compared to a tool that detects dispatches at the source code level, a tool based on binary inspection may be harder to construct, but it offers a significant advantage even beyond its source, compiler, and language independence. In particular, it is non-intrusive, i.e., does not alter the instruction sequence, and is thus more accurate.

The vf-call marker program detects the virtual function call code sequence discussed in section 2. This code sequence consists of the five instructions in Figure 2 and any intervening register moves. They may appear in different orderings (but with the correct dependencies), possibly spread out over different basic blocks. Since the code sequence is highly characteristic, the marker program is very accurate, detecting virtual calls exactly for most programs.[†] For three benchmarks the marker is slightly imprecise, erring by 0.4% or less. Only in *ixx*, 2.3% of the calls went undetected so that our measurements slightly underestimate the direct dispatch cost for this benchmark.

[†] We cross-checked this by using VPROF, a source-level virtual function profiler for GCC [Aig95].

3.2 Benchmarks

We tested a suite of two small and six large C++ applications totalling over 90,000 lines of code (Table 1). In general, we tried to obtain large, realistic applications rather than small, artificial benchmarks. Two of the benchmarks (*deltablue* and *richards*) are much smaller than the others; they are included for comparison with earlier studies (e.g., [HU94, G+95]). *Richards* is the only synthetic benchmark in our suite (i.e., the program was never used to solve any real problem). We did not yet test any programs for which only the executables were available.

name	description	lines
deltablue	incremental dataflow constraint solver	1,000
eqn	type-setting program for mathematical equations	8,300
idl	SunSoft's IDL compiler (version 1.3) using the demonstration back end which exercises the front end but produces no translated output.	13,900
ixx	IDL parser generating C++ stubs, distributed as part of the Fresco library (which is part of X11R6). Although it performs a function similar to IDL, the program was developed independently and is structured differently.	11,600
lcom	optimizing compiler for a hardware description language developed at the University of Guelph	14,100
porky	back-end optimizer that is part of the Stanford SUIF compiler system	22,900
richards	simple operating system simulator	500
troff	GNU groff version 1.09, a batch-style text formatting program	19,200

Table 1: Benchmark programs

For every program except *porky*[‡] we also tested an "all-virtual" version (indicated by "-av" suffix) which was compiled from a source in which all member functions except operators and destructors were declared virtual. We chose to include these program versions in order to simulate programming styles that extensively use abstract base classes defining virtual functions only (C++'s way of defining interfaces). For example, the Taligent CommonPoint frameworks

[‡] Porky cannot be compiled as "all virtual" without a large effort of manual function renaming.

program	version	instructions	virtual calls	instructions per virtual call
deltablue	original	40,427,339	615,100	65
	all-virtual	79,082,867	5,145,581	15
eqn	original	97,852,301	100,207	976
	all-virtual	108,213,587	1,267,344	85
idl	original	91,707,462	1,755,156	52
	all-virtual	99,531,814	3,925,959	25
ixx	original	30,018,790	101,025	297
	all-virtual	34,000,249	606,463	56
lcom	original	169,749,862	1,098,596	154
	all-virtual	175,260,461	2,311,705	75
richards	original	8,119,196	65,790	123
	all-virtual	15,506,753	1,146,217	13
troff	original	91,877,525	809,312	113
	all-virtual	114,607,159	3,323,572	34
porky	original	748,914,861	3,806,797	196

Table 2: Basic characteristics of benchmark programs (dynamic counts)

provide all functionality through virtual functions, and thus programs using CommonPoint (or similar frameworks) are likely to exhibit much higher virtual function call frequencies. Lacking real, large, freely available examples of this programming style, we created the "all virtual" programs to provide some indication of the virtual function call overhead of such programs. These versions can also be used to approximate the behavior of programs written in languages where (almost) every function is virtual, e.g., Java or Modula-3.

For each benchmark, Table 2 shows the number of executed instructions, the number of virtual function calls, and the average number of instructions between calls. All numbers are dynamic, i.e., reflect run-time execution counts unless otherwise mentioned. All programs were simulated in their entire length as shown in Table 2. Simulation consumed a total of about one CPU-year of SPARCstation-20 time.

3.3 Processors

Table 3 shows an overview of recently introduced processors. Since we could not possibly simulate all of these processors and their subtle differences, we chose to model a hypothetical SPARC-based processor that we dubbed P96 because it is meant to resemble the average processor introduced today.

Processor	Ultra SPARC	MIPS R10K	DEC Alpha 21164	Power PC 604	Intel Pentium Pro
Shipping date	95	95	95	95	95
Size of BTB	0	0	0	64	512
Size of BHT[a]	2048	512	2048	512	0
Branch Penalty	4	4	5	1-3	11-15
Issue Width	4	5	4	4	3
Load Latency		2	2	2	3
Primary I-cache[b]	16Kx2	32Kx2	8Kx1	32Kx4	8K
Primary D-cache	16Kx1	32Kx2	8Kx1	32Kx4	8K
Out-of-order ?	Y	Y	Y	Y	Y
Speculative?	Y	Y	Y	Y	Y

Table 3: Characteristics of recently introduced processors

[a] BTB = branch target buffer size; BHT = branch history table size (branch histories are used to predict the direction of conditional branches)
[b] 16Kx2 means the cache is 16K bytes , and 2-way associative

For our experiments, we ran all benchmarks on P96 to obtain the base results for the dispatch overhead. To examine the effects of the most important processor features, we then varied each parameter while keeping all others constant. Finally, we also measured a few individual configurations that resemble existing processors (Table 3). P96-noBTB resembles the UltraSPARC in that it lacks a BTB, i.e., does not predict indirect branches. P96-Pro resembles the Pentium Pro in its branch configuration, having a very high branch penalty and relatively modest branch prediction. Finally, P2000 is an idealized processor with essentially infinite hardware resources; we use it to illustrate the impact of the branch penalty on a processor that has virtually no other limitations on instruction issue.

It should be noted that none of these processors is intended to exactly model an existing processor; for example, the Intel Pentium Pro's instruction set and microarchitecture is very different from P96-Pro, and so the latter should not be used to predict the Pentium Pro's performance on C++ programs. Instead, we use these processors to mark plausible points in the design space, and their distance and relationship to illustrate particular effects or trends.

Processor	P96	P96-noBTB	P96-Pro	P2000/bp1	P2000/bp10
Size of BTB	256	0	512	1024	1024
Size of BHT	1024	1024	0	1024	1024
Branch Penalty	4	4	15	1	10
Issue Width	4	4	4	32	32
Load Latency	2				
Primary I-cache	32K, 2-way associative				
Primary D-cache	32K, 2-way associative				
Out-of-order ?	Y				
Speculative?	Y				

Table 4: Characteristics of simulated processors

4. Experimental Results

This section first examine the cost of dynamic dispatch on the baseline architecture, P96, and then examines the impact of individual architectural parameters (branch penalty/prediction, load latency, and issue width).

4.1 Direct cost on P96

4.1.1 Instructions and cycles

First, we will examine the cost of dynamic dispatch on the baseline architecture, P96. Recall that we define the cost as the additional cycles spent relative to a "perfect" dispatch implementation that implements each dispatch with a direct call. Figure 7 and Figure 8 show the results. On the standard benchmarks, the cost

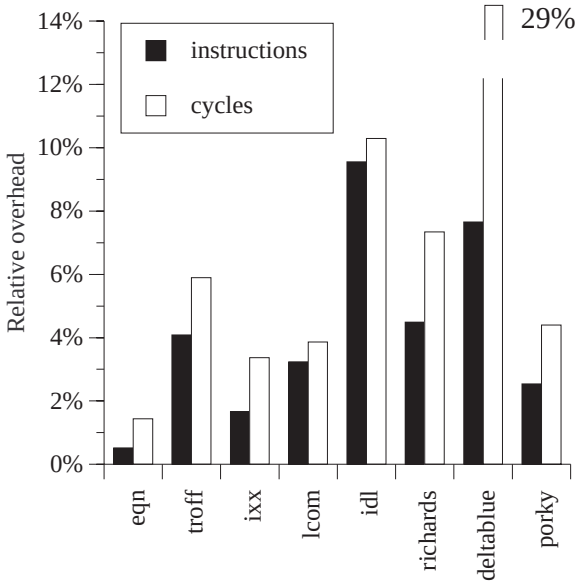


Figure 7. Direct cost of standard VFT dispatch (unmodified benchmarks)

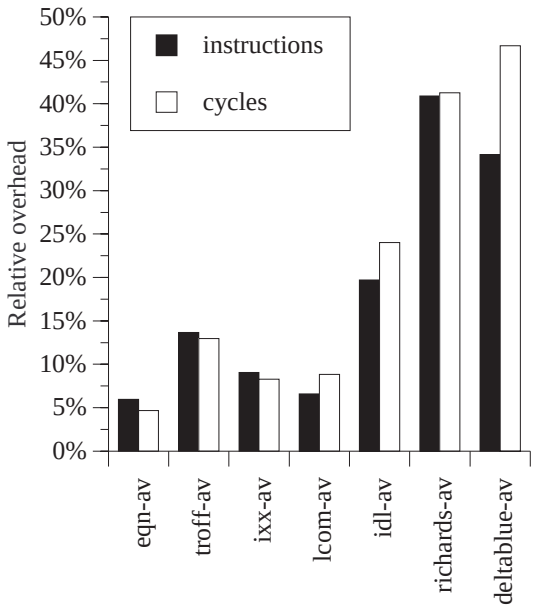


Figure 8. Direct cost of standard VFT dispatch (all-virtual benchmarks)

varies from 1.4 % for *eqn* to 29% for *deltablue*, with a median overhead of 5.2 %. For the all-virtual versions, the overhead increases to between 4.7 % and 47% with a median overhead of 13%. The standard benchmarks spend a median 3.7% of their instructions on dispatch, and the all-virtual versions a median of 13.7%. For the standard benchmarks the cycle cost is larger than the cost in the number of instructions executed; on average, it is a median 1.7 times larger. This difference confirms that the VFT dispatch sequence does not schedule well on a superscalar processor, compared to non-dispatch code. However, this effect varies substantially between benchmarks. The largest difference is found in *eqn* (2.8 times) and deltablue (3.8 times). Since the dispatch sequence is always the same, this indicates that the instructions surrounding a call can significantly affect the cost of virtual function lookup, or that virtual calls are more predictable in some programs than in others. We will explore these questions shortly.

4.1.2 Thunks

Figure 9 compares the cycle cost of standard and thunk implementations for the unmodified benchmarks[†]. Thunks have a smaller cycle overhead than regular

[†] Since GCC cannot compile *idl*, *idl-av*, and *lcom-av* with thunks, these benchmarks are missing from Figure 9 and Figure 10.

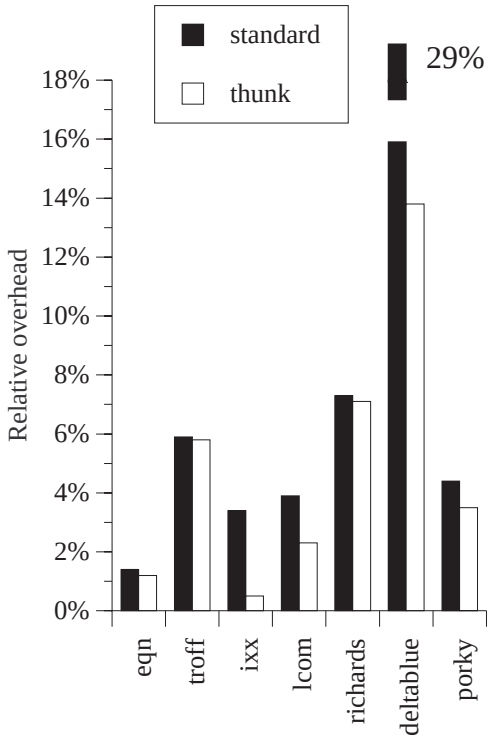


Figure 9. Cycle cost of VFT dispatch, standard and thunk variants (unmodified benchmarks)

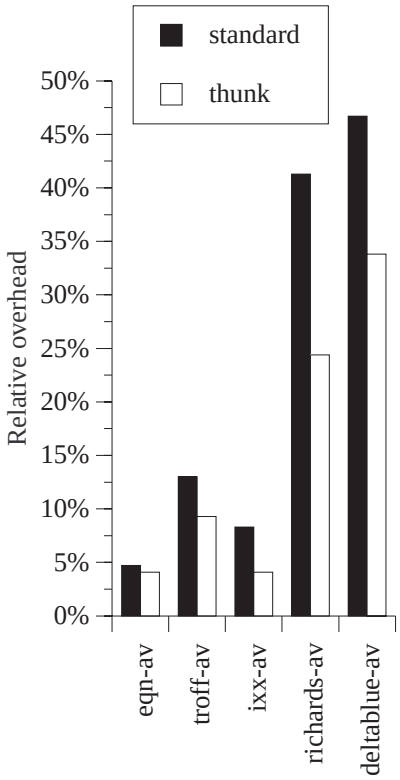


Figure 10. Cycle cost of VFT dispatch, standard and thunk variants (all-virtual benchmarks)

tables for all benchmarks, using a median of 79% of the cycles of the regular implementation. Figure 10 shows the cycle cost for the all-virtual benchmarks. Here, thunks have 72% of the regular overhead. The exact amount of the gain varies greatly between benchmarks. For example, the thunk overhead for *ixx* and *deltablue* is only 15% and 47% of the regular overhead, while for *troff*, thunks use almost as many cycles as standard tables (98%).

How can thunks, in some cases, improve dispatch performance by more than a factor of two? One reason for the difference is the unnecessary receiver address adjustment that is avoided with thunks (instructions 2 and 4 in Figure 5). In the thunk implementation, instructions that depend on the receiver's address do not have to wait for the virtual function call to complete, if the target is predicted accurately. In contrast, in the standard implementation instructions 2 and 4 create a dependency chain from instruction 1 to any instruction that needs the receiver's address. In *deltablue*, the added dependencies stretch the inner loop from 9 cycles (thunks) to 12 cycles (standard), where a direct called implementation would use 8 cycles (all times exclude cache misses). Thus the overhead of thunks is only 25% of the overhead of standard tables for a large part of the execution, so that the removal of only two instructions out of five can avoid more than half the virtual function call overhead in particular cases. This effect is particularly pronounced in all-virtual benchmarks that contain many calls to accessor functions (i.e., functions that just return an instance variable).

Another part of the difference is due to memory hierarchy effects: with perfect caching[†], thunk overhead for *ixx* and *deltablue* rises to 48% and 54%.

[†] By perfect caching we mean that there are no cache miss, not even for cold starts.

4.1.3 Generalization to other processors

How specific are these measurements to our (hypothetical) P96 processor? Figure 11 compares the relative dispatch overhead of standard tables on P96 with that of the other processors listed in Table 3. Clearly, the processor configuration affects performance: longer branch penalties combined with less ambitious branch prediction (P96-Pro) and the absence of a BTB (P96-noBTB) both impact dispatch performance negatively so that all programs spend a larger percentage of their time in dispatch code. Even P2000 with its 32-instruction issue CPU shows relative overheads that are a median 28% higher than in P96. Thus, we expect future processors to exhibit higher dispatch overheads for most C++ programs.

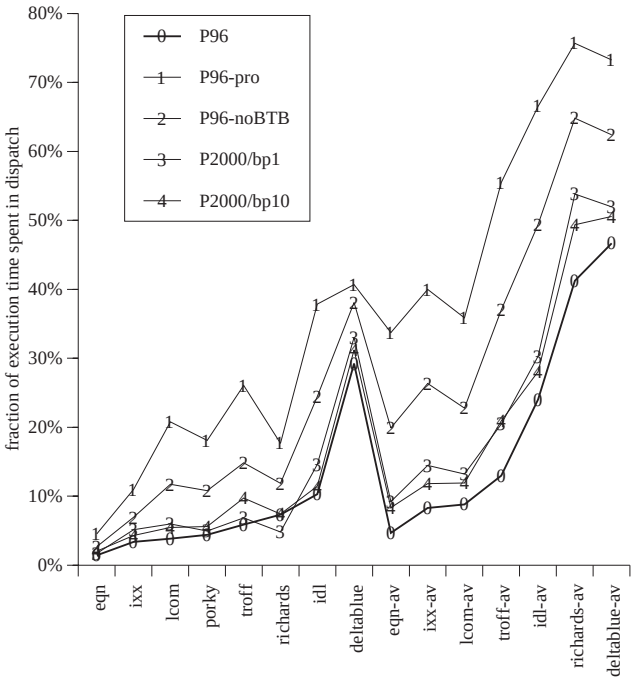


Figure 11. Dispatch overhead in P96 vs. P96-noBTB and P96-Pro

To explain these differences in more detail, the next few sections present the effects of several processor characteristics on the direct cost of dynamic dispatch. In particular, we will investigate the impact of the branch penalty, the size of the branch target buffer (BTB), and the issue width. In each experiment, we vary the feature under investigation while keeping all other characteristics constant. To illustrate the trends, we show cost in two ways, each of them relative to P96. The first graph in each section compares absolute cost, i.e., the number of dispatch cycles relative to P96. The second graph compares relative cost, i.e., the percentage of total execution time (again relative to P96) spent in dispatch. The two measurements are *not* absolutely correlated: if the absolute overhead increases, the relative cost may decrease if the rest of the application is slowed down even more than the dispatch code. Similarly, the absolute cost may decrease while the relative cost increases because the absolute cost of the rest of the application decreases even more strongly.

4.2 Influence of branch penalty

Since one of the five instructions in the dispatch sequence is an indirect branch, the branch misprediction penalty directly affects the cost of virtual function dispatch. Since each dispatch contains a single indirect branch, we would expect the absolute overhead to increase proportionally to the number of mispredicted branches. And since the number of mispredictions is independent of the branch penalty, the cost should increase linearly with the branch penalty.

name	unmodified	all-virtual
deltablue	D	d
eqn	E	e
idl	I	i
ixx	X	x
lcom	L	l
porky	P	n/a
richards	R	r
troff	T	t

Table 5: Benchmark abbreviations

Figure 12 confirms this expectation (see Table 5 for the one-letter abbreviations used in Figure 12 - 17). For small branch penalties, the actual penalty can be smaller than expected if the branch penalty is filled with instructions preceding the branch which have not yet completed (e.g. because they are waiting for their

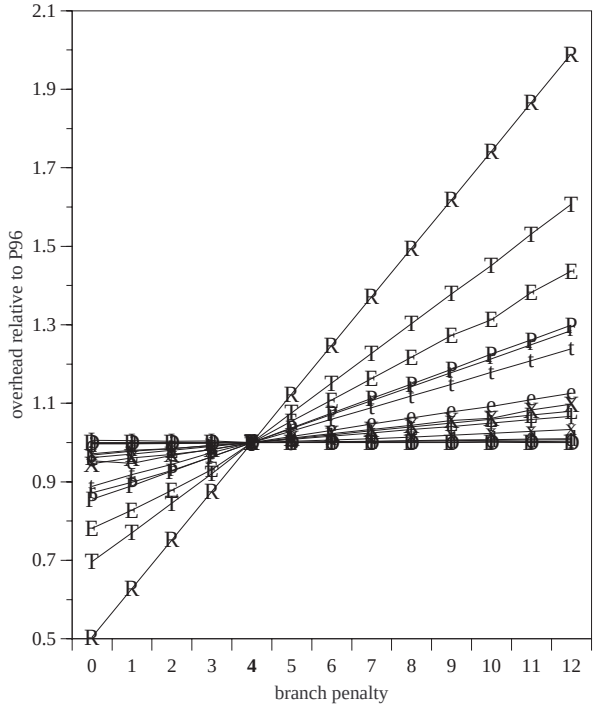


Figure 12. Overhead in cycles (relative to P96) for varying branch penalties

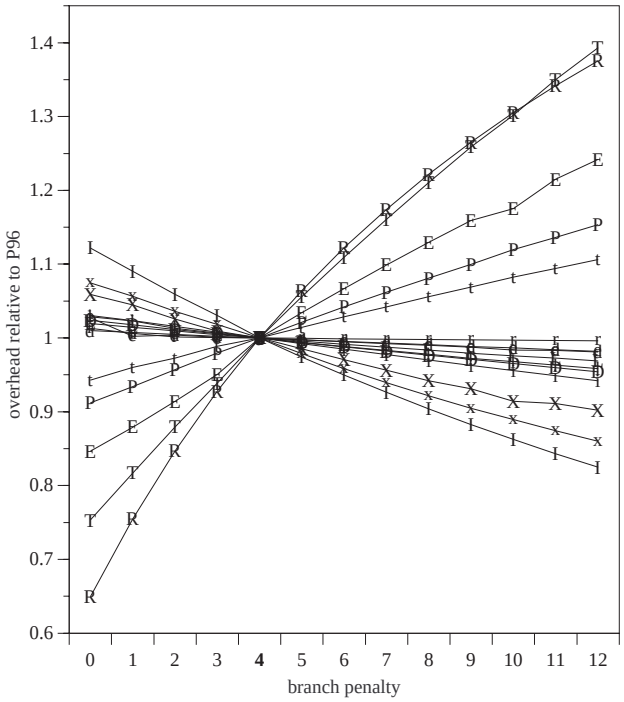


Figure 13. Overhead in % of total execution time (relative to P96) for varying branch penalties

inputs to become available). This effect appears to be small.

The slope of the overhead lines increases with the BTB miss ratio, i.e., the fraction of mispredicted calls. *Richards* and *troff* have large BTB miss ratios (54% and 30%), which account for their steep cost curves. Most of the other benchmarks have a misprediction rate of 10% or less, which dampens the effect of branch penalty on cycle cost.

Figure 13 shows that the *fraction* of execution time spent in dispatch can actually decrease with increasing branch penalty. For example, *ixx* has many indirect calls that are not part of virtual function calls, and these branches are very unpredictable (with a BTB miss ratio of 86%). Consequently, the relative overhead of virtual calls in *ixx* decreases with larger branch penalties since the cost of the rest of the program increases much faster.

However, for most benchmarks the relative overhead differs less than 20% between the extreme branch penalty values (0 and 10), indicating that the VFT branches are about as predictable as the other branches in the applications. Thus, the relative dispatch costs given earlier in Figure 7 and Figure 8 are quite insensitive to branch penalty variations.

4.3 Influence of branch prediction

As discussed in section 2.3, branch target buffers (BTBs) predict indirect (or conditional) branches by storing the target address of the branch's previous execution. How effective is this branch prediction? Our baseline processor, P96, has separate prediction mechanisms for conditional and indirect branches since the former can better be predicted with history-sensitive 2-bit predictors [HP95]. Thus, varying the size of the BTB will affect only indirect branches, thus directly illustrating the BTB's effect on dispatch overhead.

In general, smaller BTBs have lower prediction ratios because they cannot store as many individual branches. Recall that the processor uses the branch instruction's address to access the BTB (just like a load instruction uses the data address to access the data cache). If the branch isn't cached in the BTB, it cannot be predicted.

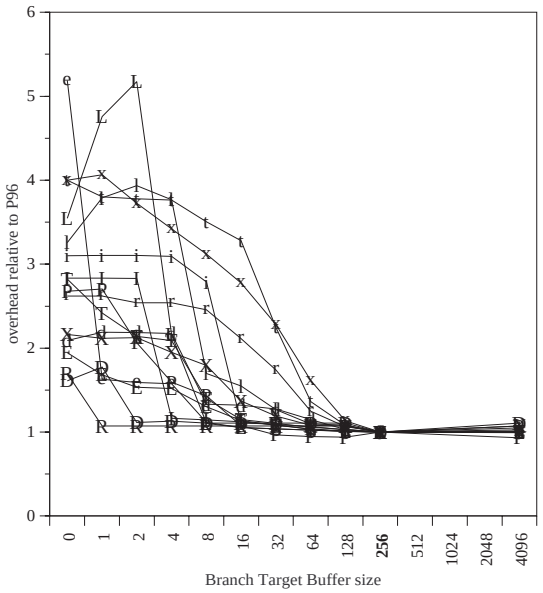


Figure 14. Overhead in cycles (relative to P96) for varying Branch Target Buffer sizes

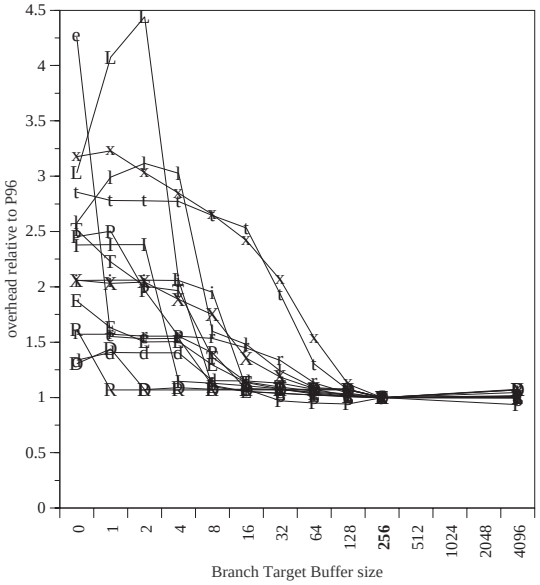


Figure 15. Overhead in % of total execution time (relative to P96) for varying Branch Target Buffer sizes

Naturally, the smaller the BTB, the fewer branches it can hold, and thus the larger the fraction of branches that can't be predicted because they aren't currently cached in the BTB. Figure 14 confirms this expectation: in general, smaller BTBs increase dispatch overhead.† Apparently, a BTB size of 128 entries is large enough to effectively cache all important branches, as the dispatch overhead does not decrease significantly beyond that BTB size.

Figure 15 shows the dispatch overhead as a fraction of execution time. In general, the relative overhead varies in tandem with the absolute overhead, i.e., smaller BTBs increase dispatch overhead. For processors with BTBs with 128 or more entries, P96 should accurately predict the BTB's impact on dispatch performance.

Finally, Figure 16 shows the prediction ratio as a function of the BTB size. The ratio starts at zero (without a BTB, indirect branches cannot be predicted) and asymptotically reaches a final value around a BTB size of 128. Generally, smaller benchmarks need fewer BTB entries to reach asymptotic behavior since they have fewer active call sites.

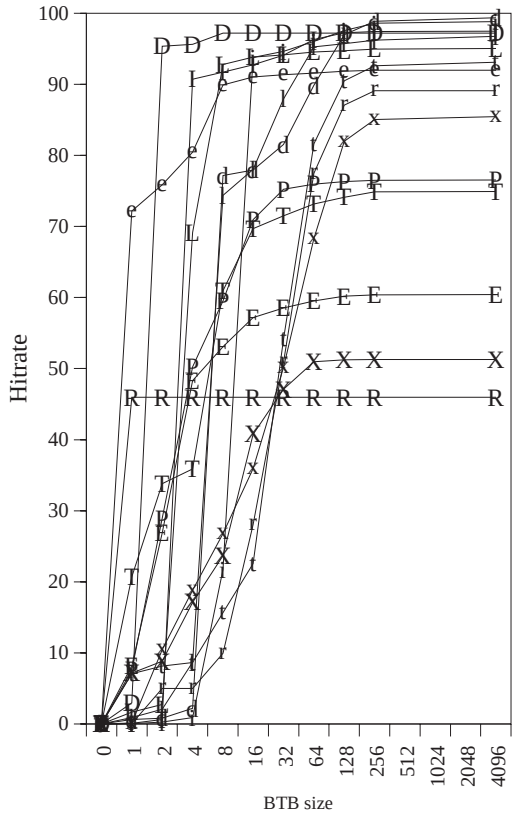


Figure 16. Indirect branch prediction ratio as a function of BTB size

† For very small BTB sizes, the overhead can *increase* with a larger BTB. This is not a bug in our data. In very small BTBs, there are many conflict misses—branches are evicting each other from the BTB because the BTB cannot cache the working set of branches. With very small BTBs, this thrashing is so bad that removing the virtual calls does not improve hit ratios in P_{ideal} . However, at some point the BTB may be just large enough to hold most indirect branches in P_{ideal} but still not large enough to also hold the virtual function calls. In this case, the difference in BTB effectiveness between P_{ideal} and P suddenly becomes large, thus leading to a higher dispatch overhead.

The asymptotic prediction ratio corresponds to the hit ratio of an inline cache[†] [DS84]. For some benchmarks, prediction works very well, with 90% or more of the calls predicted correctly. But several benchmarks (especially *richards*, *ixx*, *eqn*, and *troff*) show much lower prediction ratios even with very large BTBs because their calls change targets too frequently. For example, the single virtual call in *richards* frequently switches between four different receiver classes, each of which redefines the virtual function. No matter how large the BTB, such calls cannot be predicted well. The median prediction ratio for the standard benchmarks is only 65% vs. 91% for the all-virtual versions; the latter are more predictable because many calls only have a single target and thus are predicted 100% correctly after the first call.

4.4 Influence of load latency

Load latency influences dispatch cost since the VFT dispatch sequence contains two dependent load instructions. Thus, higher load latencies should lead to higher dispatch overhead. Our measurements confirm this assumption: compared to the baseline load latency of two, increasing the load latency to three increases absolute dispatch cost by a median of 51%; the relative cost increases by 31%. Similarly, with a load latency of one the absolute overhead decreases by 44% and the relative overhead by 37%. (Processors are unlikely to have load latencies larger than three, so we did not simulate these.)

Clearly, load latency affects the efficiency of dispatch code more than that of "normal" code sequences. Furthermore, it appears that there are not enough surrounding application instructions to effectively hide the latency of the loads in the dispatch sequence, even for small load latencies.

4.5 Influence of issue width

The final factor, issue width (i.e., the number of instructions that can be issued to the functional units in one cycle) can also influence dispatch performance. Figure 17 shows that issue width has a strong impact for small values. On a scalar processor (issuing at most one instruction per cycle), programs spend a much smaller fraction of their time in dispatch. Of course, absolute performance would be worse than on P96 since execution would consume many more cycles (for example, *lcom* is three times slower on the one-issue processor than on the four-issue processor). With larger issue widths the relative overhead increases more slowly, reaching an asymptotic value of 26% (median) more than on P96. Thus, on wider-issue processors, the relative cost of dynamic dispatch will increase slightly because the application code benefits more from the additional issue opportunities than the dispatch code.[‡]

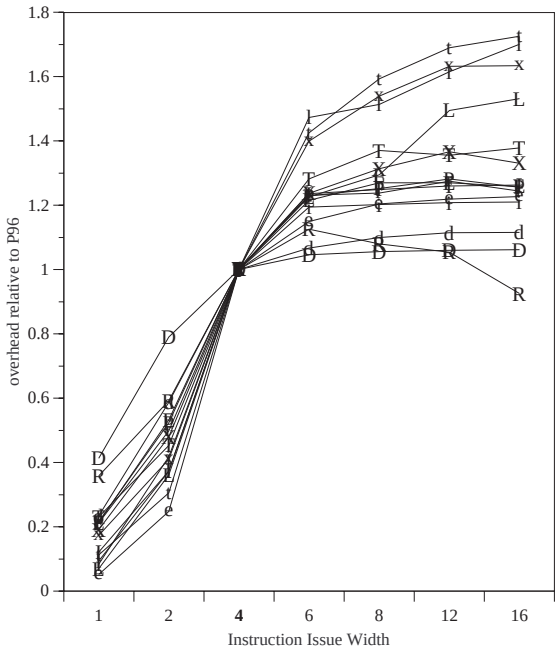


Figure 17. Overhead in % of total execution time (relative to P96) for varying instruction issue widths

4.6 Cost per dispatch

In [DHV95] we predicted the cost of a single VFT dispatch to be 2L + B + 1, i.e., two load delays plus a branch penalty; for P96, this adds up to 9 cycles. How accurate is this prediction? Figure 18 shows the cost in cycles per dispatch for all benchmarks. Clearly, the cost estimate of 9 cycles is too high, but that is not

[†] Since an inline cache stores a target separately for each call site, its hit rate mirrors that of a branch target buffer of infinite size with no history prediction bits.

[‡] For a few benchmarks (e.g., *richards*) the relative overhead decreases with high issue widths. We assume that these benchmarks benefit from higher issue rates because they allow critical dispatch instructions to start earlier, thus hiding part of their latency.

surprising because the above model ignores the effects of branch prediction and co-scheduling of non-dispatch instructions. In essence, a BTB reduces the effective branch penalty since the full penalty B is only incurred upon a misprediction. The cost model could be improved by using the effective branch penalty B_{eff} = B * btb_misprediction_ratio. For the standard benchmarks, with a median misprediction ratio of 35%, this model predicts a cost of 6.4 cycles, which still overestimates the real cost (median 3.9 cycles). Considering all benchmarks, the median misprediction ratio of 11% results in an estimated cost of 5.4 cycles per dispatch, which overestimates the actual median of 2.8 cycles / dispatch by a factor of two.

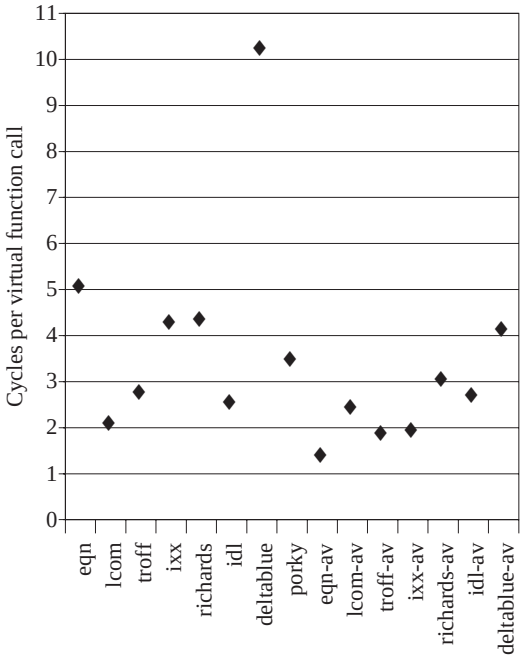


Figure 18. Cycles per dispatch

Dispatch cost varies widely: a single dispatch costs 2.1 cycles in *lcom* but 10.2 cycles in *deltablue*, a difference of a factor of 4.8. This variation illustrates the combined effects of the factors discussed previously, such as the BTB hit ratio and the co-scheduling of application code. The dispatch cost of the all-virtual programs varies much less since the average cost is dominated by very predictable monomorphic calls (i.e., call sites invoking the same function every time).

5. Discussion and Future Work

What does this detailed dispatch performance analysis tell us? Will dispatch performance improve with future hardware? Should programmers write their applications differently to improve performance?

First, the median dispatch overheads we observed (5.2% for the standard benchmarks and 13.7% for the all-virtual versions) can be used as a bound on the dispatch performance improvements one can hope to obtain, for C++ programs, from better software or hardware. Thus, no matter how good a dispatch mechanism is, we cannot hope for much more than a performance improvement of around 5-10%. Any further improvement must come from other optimizations such as customization or inlining [CUL89, HU94]. Given that better optimizing compilers are possible [AH96], it hardly seems appropriate for programmers to compromise the structure of their programs to avoid dispatch.

Many object-oriented systems use or could use VFT-like dispatch mechanisms (e.g., implementations of Java, Modula-3, Oberon-2, and Simula), and thus this study bears some significance for those languages as well. While the characteristics of typical programs may differ from the C++ programs measured here, the general trends should be similar. Together, the standard and all-virtual programs represent a wide spectrum of program behaviors and call frequencies, and thus we expect many programs written in other languages to fall somewhere within that spectrum.

Furthermore, the dependency structure (and thus performance on superscalar processors) of many other dispatch mechanisms (e.g., selector coloring or row displacement) is similar to VFT, as we have shown in [DHV95]. Therefore, the measurements presented here should apply to these dispatch mechanisms as well.

Although simulations provide accurate numbers, they are inordinately expensive and complicated. As discussed in section 4.6, the analytical model for VFT dispatch cost developed in [DHV95] already predicts dispatch cost fairly well using only two parameters. In future work, we intend to use the detailed results presented here as a starting point to construct a better

model that would allow implementors or programmers to estimate the dispatch cost in their application using a simple formula containing few processor- or application-specific parameters.

Our results show that there may be room for better dispatch algorithms: a 5% or 10% improvement in performance for most programs is still significant. We hope that our measurements encourage others to search for better dispatch mechanisms. Previous work suggests that inline caching [DS84] should perform very well on superscalar processors [DHV95], at least for call sites with low degrees of polymorphism. In essence, inline caching is the software equivalent of an infinite branch target buffer (BTB) since it caches the last dispatch target by modifying the call. In addition, it contains only a single data dependency and thus schedules very well [DHV95]. A hybrid, adaptive dispatch implementation that employs inline caching where appropriate might considerably reduce dispatch cost in many programs and thus appears to be an attractive area for future work.

Finally, will dispatch overhead increase in the future? We believe so, even though the effect is likely to be moderate. As Figure 17 showed, the relative overhead will increase as processors issue more instructions per cycle. At an issue width of 16, the median overhead increases by about 26%. Future processors might also have longer load latencies, further increasing dispatch cost. General compiler optimizations may also influence dispatch performance. Much current research focuses on compilation techniques to increase instruction-level parallelism. If compilers successfully reduce execution time on wide-issue processors, the effective dispatch overhead could further increase for programs with unpredictable VFT calls. In summary, over the next few years, we expect the relative dispatch cost to rise, though the exact extent is hard to predict.

6. Related Work

Rose [Ros88] analyzes dispatch performance for a number of table-based techniques, assuming a RISC architecture and a scalar processor. The study considers some architecture-related performance aspects such as the limited range of immediates in instructions. Milton and Schmidt [MS94] compare the performance of VTBL-like techniques for Sather. Neither of these studies take superscalar processors into account.

The efficiency of message lookups has long been a concern to implementors of dynamically-typed, pure languages like Smalltalk where dispatches are more frequent since these languages model even basic types like integers or arrays as objects. Dispatch consumed a significant fraction of execution time in early Smalltalk implementations (often 30% or more, even in interpreted systems). Hash tables reduced this overhead to around 5% [CPL83]; however, 5% of a relatively slow interpreter still is a lot of time. The introduction of *inline caching* [DS84, UP87] dramatically diminished this overhead by reducing the common case to a comparison and a direct call. A variant, polymorphic inline caches (PICs), extends the technique to cache multiple targets per call site [HCU91]. For SELF-93 which uses inline caching and PICs, Hölzle and Ungar [HU95] report an average dispatch overhead of 10-15% on a scalar SPARCstation-2 processor, almost half of which (6.4%) is for inlined tag tests implementing generic integer arithmetic. (This figure also includes other inlined type tests, not just dispatched calls.) Given the large differences in languages, implementation techniques, and experimental setup, used, it is difficult to compare these results with those presented here.

Calder et al. [CG94] discuss branch misprediction penalties for indirect function calls in C++. Based on measurements of seven C++ programs, they conclude that branch target buffers are effective for many C++ programs. For their suite of programs (which differs from ours), they measured an average BTB hit ratio of 91%, assuming an infinite BTB. In comparison, the hit ratios we observed were much lower, with a median hit ratio of only 65% for the standard benchmarks. Grove et al. [G+95] also report more polymorphic C++ programs than Calder, which leads us to believe that Calder's suite of C++ programs may have been uncharacteristically predictable.

Srinivasan and Sweeney [SS95] measure the number of dispatch instructions in C++ applications, but do not

calculate the relative dispatch overhead or consider superscalar issue.

Much previous work has sought to improve performance by eliminating dispatches with various forms of inlining based on static analysis or profile information. Hölzle and Ungar [HU94] estimate that the resulting speedup in SELF is five times higher than the direct cost of the eliminated dispatches. Given the dispatch overheads reported here, this ratio suggests significant optimization opportunities for C++ programs. Preliminary results from an optimizing C++ compiler confirm this assumption [AH96].

7. Conclusions

We have analyzed the direct dispatch overhead of the standard virtual function table (VFT) dispatch on a suite of C++ applications with a combination of executable inspection and processor simulation. Simulation allows us to precisely define dispatch overhead as the overhead over an ideal dispatch implementation using direct calls only. On average, dispatch overhead is significant: on a processor resembling current superscalar designs, programs spend a median overhead of 5.2% and a maximum of 29% executing dispatch code. However, many of these benchmarks use virtual function calls quite sparingly and thus might underrepresent the actual "average" C++ program. For versions of the programs where every function was converted to a virtual function to simulate programming styles that extensively use abstract base classes defining virtual functions only (C++'s way of defining interfaces), the median overhead rose to 13.7% and the maximum to 47%. On future processors, this dispatch overhead is likely to increase moderately.

On average, thunks remove a fourth of the overhead associated with the standard implementation of virtual function calls. For some programs the difference is much higher since thunks remove a data dependency chain that inhibits instruction level parallelism.

To our knowledge, this study is the first one to quantify the direct overhead of dispatch in C++ programs, and the first to quantify superscalar effects experimentally. In addition to measuring bottom-line overhead numbers, we have also investigated the influence of specific processor features. Although these features typically influence the absolute dispatch cost considerably (i.e., the number of cycles spent in dispatch code), the relative cost (the percentage of total execution time spent in dispatch code) remains fairly constant for most parameters except for extreme values. Thus, the overheads measured here should predict the actual overhead on many current processors reasonably well.

Since many object-oriented languages use virtual function tables for dispatch, and since several other dispatch techniques have identical execution characteristics on superscalar processors, we believe that our study applies to these languages as well, especially if their application characteristics fall within the range of programs studied here.

Acknowledgments

This work is supported in part by NSF grant CCR 96-24458, MICRO grant 95-077, IBM Corporation, and Sun Microsystems. We would like to thank David Bacon, Harini Srinivasan, Ole Agesen, and Gerald Aigner for their comments on earlier versions of this paper. Special thanks go to Kathryn O'Brien for her support. Many thanks also go to the users of the CS department's 64-processor Meiko CS-2 (acquired under NSF grant CDA 92-16202) who accommodated our (at times extensive) simulation workload.

8. References

[Aig95] Gerald Aigner. *VPROF: A Virtual Function Call Profiler for C++*. Unpublished manuscript, 1995.

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. *ECOOP '96 Conference Proceedings*, Linz, Austria, July 1996.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 397-408, January 1994.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October 1989.

[CK93] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Sun Microsystems Laboratories, Technical Report SMLI TR-93-12, 1993. Also published as Technical Report CSE-TR 93-06-06, University of Washington, 1993.

[CPL83] T. Conroy and E. Pelegri-Llopart. An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In [Kra83].

[DM73] O.-J. Dahl and B. Myrhaug. *Simula Implementation Guide*. Publication S 47, NCC, March 1973.

[DS84] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Modern Computer Architectures. *ECOOP '95 Conference Proceedings*, Århus, Denmark, August 1995.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[G+95] David Grove, Jeffrey Dean, Charles D. Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA'95, Object-Oriented Programming Systems, Languages and Applications*, p. 108-123, Austin, TX, October 1995.

[HP95] Hennessy and Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1995.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP'91 Conference Proceedings*, Geneva, 1991. Published as *Springer Verlag Lecture Notes in Computer Science 512*, Springer Verlag, Berlin, 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing Dynamically-dispatched Calls With Run-Time Type Feedback. In *PLDI '94 Conference Proceedings*, pp. 326-335, Orlando, FL, June 1994. Published as *SIGPLAN Notices 29(6)*, June 1994.

[HU95] Urs Hölzle and David Ungar. Do Object-Oriented Languages Need Special Hardware Support? *ECOOP '95 Conference Proceedings*, Århus, Denmark, August 1995.

[Kra83] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[Kro85] Stein Krogdahl. Multiple Inheritance in Simula-like Languages. *BIT* 25, pp. 318-326, 1985.

[LS95] James Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *PLDI '95 Conference Proceedings*, pp. 291-300, La Jolla, CA, June 1995. Published as *SIGPLAN Notices 30(6)*, June 1995.

[Mic94] Microprocessor Report. *HP PA8000 Combines Complexity and Speed*. Volume 8, Number 15, November 14, 1994.

[Mic95] Microprocessor Report. *Intel's P6 Uses Decoupled Superscalar Design*. Volume 9, Number 2, February 16, 1995.

[MS94] S. Milton and Heinz W. Schmidt. *Dynamic Dispatch in Object-Oriented Languages*. Technical Report TR-CS-94-02, The Australian National University, Canberra, January 1994.

[Ros88] John Rose. Fast Dispatch Mechanisms for Stock Hardware. *OOPSLA'88 Conference Proceedings*, p. 27-35, San Diego, CA, November 1988. Published as *SIGPLAN Notices 23(11)*, November 1988.

[SS95] Harini Srinivasan and Peter Sweeney. *Evaluating Virtual Dispatch Mechanisms for C++*. IBM Technical Report RC 20330, Thomas J. Watson Research Laboratory, December 1995.

[UP87] David Ungar and David Patterson. What Price Smalltalk? In *IEEE Computer* 20(1), January 1987.