August 30, 2000

SRC Research Report

166

Efficient and Flexible Value Sampling

Mike Burrows
Ulfar Erlingson
Shun-Tak A. Leung
Mark T. Vandevoorde
Carl A. Waldspurger
Kevin Walker
William E. Weihl

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Efficient and Flexible Value Sampling

Mike Burrows, Ulfar Erlingson, Shun-Tak A. Leung, Mark T. Vandevoorde, Carl A. Waldspurger, Kevin Walker, William E. Weihl

August 30, 2000

Publication History

A version of this report will appear in the proceedings of The Ninth International Conference on Achitectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA. Republished by permission.

Ulfar Erlingson is currently working at deCODE Genetics.
Mark Vandevoorde is currently working at the AltaVista Company.
Carl Waldspurger is currently working at VMware, Inc.
Kevin Walker is currently working at SiByte, Inc.
Bill Weihl is currently working at Akamai Technologies, Inc.

Abstract

This paper presents novel sampling-based techniques for collecting statistical profiles of register contents, data values, and other information associated with instructions, such as memory latencies. Values of interest are sampled in response to periodic interrupts. The resulting value profiles can be analyzed by programmers and optimizers to improve the performance of production uniprocessor and multiprocessor systems.

Our value sampling system extends the DCPI continuous profiling infrastructure, and inherits many of its desirable properties: our value profiler has low overhead (approximately 10% slowdown); it profiles all the code in the system, including the operating system kernel; and it operates transparently, without requiring any modifications to the profiled code.

1 Introduction

Hardware-based value prediction mechanisms were originally proposed by Lipasti and Shen [13] to reduce pipeline delays for long-latency operations. Simulations indicated a surprising amount of locality in the values computed by instructions, allowing some result values to be predicted accurately based on prior executions of the same instruction.

Software-based value profiling was first investigated by Calder, Feller and Eustace [4, 5, 9]. A value profiler records values generated by the instructions in a program, and maintains statistics about the observed values. For example, a value profiler might report that, 53% of the time, the instruction at PC 0x2468 generates the result value 0, and the rest of the time its result value is 1.

There are several possible approaches to implementing a value profiler. A binary-rewriting tool can be used to *instrument* a program, adding code to capture the results generated by instructions; Calder *et al.* used ATOM [16] to instrument binaries. Alternatively, a machine simulator or emulator can be modified to record values of interest during *simulation*. This was the approach used in various architectural studies of value prediction. Finally, timer-based interrupts can be employed to periodically *sample* values as a program executes. We pursued this last technique, which we refer to as *value sampling* when we wish to distinguish it from the other approaches.

We generalize the traditional notion of value profiling by allowing users to capture a wide variety of values associated with the execution of the code. For example, in addition to recording values generated by the program being profiled, we might also collect timing information (*e.g.*, this load took 20ns), as well as state not directly visible to the running program (*e.g.*, this load hit in the second-level cache; the physical address accessed by this store was 0x561c).

Value profiling has a number of practical uses. It can provide data for evaluating proposed hardware features [13]. Value profiles provide feedback that can help focus manual tuning or drive automated optimizations [5]. They can also be used in debugging, although we currently have little experience with this application.

Several code optimizations are enabled when a value profile reveals places where values are *invariant* (or *semi-invariant* [4])—that is, places where some variable or register always (or almost always) contains the same value. Such optimizations include:

- **Prefetching:** a value profile can reveal which addresses are accessed, and identify absolute addresses or relative offsets that are highly predictable.
- **Specialization:** a value profile can identify common values of procedure arguments, allowing significantly better code generation. For example, at a

given call site, the *log()* routine may always be called with the argument 1.0, which admits a particularly fast implementation. Similarly, virtual method calls in object-oriented languages can be specialized for their most common receiver classes.

- **Speculation:** a value profile can expose opportunities for software speculation, allowing predicted values to be used for dependent instructions while the actual values resulting from long-latency operations are still being computed. Such optimizations might be particularly effective on architectures that support predicated execution, such as IA-64.

Value profiles can highlight the reasons why a piece of code is performing poorly, allowing tuning effort to be focused more effectively. For example, by observing load latency information, a programmer might realize that a data structure is being shared between processors in an inefficient way.

Our value sampling system extends the Digital Continuous Profiling Infrastructure (DCPI) [2], which we briefly review here. DCPI is a profiler based on statistical sampling, combined with a set of profile analysis tools. DCPI uses frequent randomized periodic interrupts to obtain samples across almost all code running on the machine, including the operating system kernel. Each DCPI sample contains a PC and address space identifier, and may optionally include information about other events (such as cache misses or branch mispredictions) depending on the specific processor implementation [2, 7]. A device driver aggregates samples and passes them on to a user-space daemon process. The daemon uses information from the dynamic linker and the operating system to map address space identifiers to object files (executable and libraries) in the file system, and stores the samples in files grouped according to the object files they refer to. Analysis tools use the samples in various ways, from providing traditional CPU time profiles of procedures, to inferring the reasons for dynamic pipeline stalls at individual instructions. Careful implementation yields an overall overhead of a few percent, despite a sampling interval of about 64 thousand instructions.

By building on DCPI, we inherit its overall structure of a kernel device driver, a user-space daemon, and analysis tools that access profiles via the file system. We also inherit a number of DCPI's advantages:

- **Efficiency:** Periodic sampling can have dramatically less overhead than value profiling schemes based on binary modification or interpretation. When we apply value sampling to all address spaces, we see overheads around 10%, using the same sampling intervals normally used by DCPI. This overhead compares favorably with the order-of-magnitude slowdowns reported for value profiling systems based on binary instrumentation.

- **Completeness:** We are able to apply value sampling to the operating system kernel, and other privileged address spaces that would be difficult to handle by other means.

- **Transparency:** Programs are slowed down slightly, but otherwise unaffected by being profiled. There is no danger of unexpected interactions arising from the use of per-address-space resources (*e.g.*, virtual addresses, or file descriptors).

Similarly, we inherit DCPI's primary disadvantage: It is a sampling-based approach, and so cannot capture *all* values observed in a run of a program. However, in practice we have not found this to be a problem.

In the following sections, we provide details of our implementation, our experiences using it, and what we believe we have learned.

2 Our Approach

Our value sampling system extends DCPI in three key ways. First, the interrupt routine captures values from the interrupted program, in addition to the usual PC samples and event records. Second, to limit the space needed to hold value samples, we employ further sampling techniques described by Gibbons and Matias [12]. These allow us to maintain efficiently *hotlists* containing the most frequently seen values at each PC, using constant space per hotlist. Finally, we have developed additional analysis tools to process the value samples. For example, the user can display values observed together with their associated assembly-language instructions and higher-level source statements. Other tools automatically find semi-invariant values in code that is being executed a significant number of times.

2.1 Gathering Value Samples

We use the performance counters available on Alpha processors to interrupt each running CPU periodically. At each interrupt, we record values from the current context. Typically, the sampling interval is 64 thousand instructions, though a small amount of randomization is added to avoid unwanted timing interactions.

The first question in such a system is how to obtain data values from an interrupted context. Without knowledge of the path followed by the processor's PC just prior to the interrupt, one cannot trivially associate the values in the registers with particular instructions.

2.1.1 An Early Attempt

Our first attempt at solving this problem was a "bounce back" technique that arranges for a second performance counter interrupt to occur after a small number of instructions (such as one issue block) has been executed immediately after resuming the original interrupted code. During the first "setup" interrupt, the return PC and other instructions in its issue block are fetched and recorded to determine which registers will contain values of interest. During the second "bounce back" interrupt, the values of interest (register values, return address, etc.) are captured and recorded.

Ensuring that exactly one issue block is executed between the two interrupts proved fairly difficult because a large number of kernel instructions are executed in the interrupt return path. We were assisted by a feature of the Alpha 21164 CPU, which can generate an interrupt after a specified number of cycles in user-mode. We were able to make the delivery of this interrupt fairly predictable by evicting the i-cache line containing the issue block of interest, and taking the i-cache fill time into account. Nevertheless, we would sometimes observe that no progress had been made in user mode before the second interrupt was delivered. In such a case, we would increase the number of user-mode cycles that would trigger the "bounce back" interrupt. If too much progress was made, we would give up our attempt to collect data at this interrupt—this happened on a few percent of interrupts. A rarer problem was that the amount of progress made between two interrupts was sometimes ambiguous because of tight loops in the interrupted code.

Although we successfully prototyped the "bounce back" mechanism, it worked only on user-mode code and only with some Alpha processors (the 21164 family). In the light of these limitations, we sought an alternative.

2.1.2 Using an Interpreter

Ultimately, we added a complete interpreter for the Alpha instruction set to the DCPI kernel module. The interrupt routine interprets the next several instructions, advancing the interrupted context as though those instructions had been executed directly by the processor. Though conceptually simple, there are some practical concerns with this approach.

First, the interpreter must be correct and reasonably complete. Although the interpreter can give up if it should encounter an instruction it cannot handle, it is important that such instructions be rare or the profiling will have significant blind spots. Thus we handle the entire instruction set, and rigorous testing was used to gain confidence in the interpreter.

One might think that we could have run the interpreter without having side-effects on the interrupted context, and this would relax the need for correctness in the interpreter. An error in the interpreter might produce erroneous value profiles, but would not affect the profiled program. We dismissed this approach because we wished to apply value sampling to the operating system kernel, which performs loads on device registers that may have side-effects.

No matter how complete the interpreter, there are still coverage limitations. We are unable to apply it to code where no interrupts are permitted, such as Alpha's PAL code [1] and certain small parts of the kernel. Some operations cannot easily be emulated by the interpreter because it is running at high interrupt level. In particular, the interpreter gives up when it encounters any of the following:

- traps, such as page faults, that cannot be handled at high interrupt levels;
- a change to the interrupt level; and
- a change to the kernel stack pointer—the interpreter is using the same stack.

The interpreter provides flexibility not available through the earlier "bounce back" scheme. In particular, the interpreter can be modified to record timing information for individual long-running instructions such as loads; this will be discussed in Section 3.3. Similarly, the interpreter could record other system state, such as page table contents, or the interrupt level in the interrupted context. Or it can be modified to simulate some internal state of a particular processor in order to deduce where the processor might perform poorly. Quite complex analysis can be performed in the interrupt routine, provided that time critical interrupts are not masked for too long.

2.1.3 User-Mode Interpretation

We also support an alternative means for invoking the interpreter, which has a different set of advantages and disadvantages. Instead of running the interpreter in the interrupt routine, we are able to run it as a user-mode library in the profiled address space, using an upcall mechanism.

When the address space is created, the dynamic linker loads a value-profiling shared library along with the application. The library registers the address space with the profiling driver. At each profiling interrupt, the driver revectors the user-mode context to the library's user-mode trap handler that runs the interpreter, logs the data obtained, and finally returns control to the interrupted context. This is similar to the intended use of the SIGPROF signal in some UNIX systems.

The user-mode approach has different practical implications:

- The address space is being disturbed in ways other than timing—a new shared library is being loaded, and new code is being run on the user-mode thread stacks.

- The interpreter does not run at high interrupt level, so there is no limit on the amount of time that can be spent in the interpreter.

- Page faults encountered by the interpreter will be resolved by the operating system in the normal way, so interpretation will not cease at page faults.

- Some values available in the kernel, such as physical addresses, will not be available directly to user mode. Similarly, some data may be easier to obtain in user-mode, such as data revealed from a stack trace of the interrupted context.

- The correctness of the interpreter affects only a single address space, so in principle users could modify the interpreter to collect specialized information.

- The user-mode approach makes it straightforward to perform value sampling in interpreted languages.

We expect that some users will prefer to run the value-profiling interpreter in user mode, while others will want to run it in the kernel.

2.2 Data Reduction

Given a basic mechanism for capturing values, a second problem is that of data reduction. The number of values observed at any given point in the program might be very large—far too large to store conveniently. Calder, Feller, and Eustace [4, 5] employed a small table to hold the most frequently seen values. However, their *ad hoc* update policy required tuning to get good results.

We used Gibbons and Matias' techniques [12] for summarizing a stream of data. These techniques provide a statistically sound basis for keeping a list of the most-frequently-seen values in a stream of values; their main advantage over an *ad hoc* scheme is that no tuning is required, and they use less memory for a given result quality.

We briefly describe the simplest scheme for keeping track of the top N most frequently seen values in a data stream; for more details we recommend Gibbons and Matias' paper. Conceptually, the algorithm keeps a probability p and a table C that maps each possible value v to a counter $C[v]$. The algorithm maintains the invariant that $C[v]/p$ is an unbiased estimate of the number of times v has been seen in the data stream. Let $NZ(C)$ be the number of non-zero counters in C. Initially, $p = 1$ and $\forall v : C[v] = 0$, so $NZ(C) = 0$. The table C has space for at most N values with non-zero counters; that is, $NZ(C) \leq N$. For each v in the data stream, one is added to counter $C[v]$ with probability p. If that causes $NZ(C)$ temporarily to exceed N, the following operation is repeated until $NZ(C) \leq N$ once more: For some arbitrary value $f > 1$, p is reduced to p/f, and each value instance recorded in C is retained with probability $1/f$. That is, each non-zero $C[v]$ is replaced by the number of heads seen when tossing $C[v]$ biased coins, where the probability of heads is $1/f$. A typical value for f is $N/(N-1)$.

We chose to keep track of the 16 most-frequently-seen values captured at each program location. That is, we run one instance of Gibbons and Matias' algorithm with $N = 16$ and $f = 16/15$ for each value type captured at each program location.

2.3 Interesting Values

The value sampling system could capture many different values associated with the interrupted context, beyond those generated directly by the programmed instructions. We have implemented a few:

- Stack context information, such as the current procedure's return address.
- Latencies for long-running instructions, such as memory accesses. This is measured when the instruction is interpreted by surrounding the operation by reads of a cycle counter.

Other possibilities are:

- Processor or hardware state, such as the current physical processor or processor set, physical addresses associated with memory accesses, and the processor interrupt priority level. Similarly, the interpreter can simulate execution of instructions for a processor pipeline or memory system that does not exist, and capture relevant internal state.
- OS or runtime system state, including various identifiers (current process, parent process, user, group, and controlling tty), privilege level (*e.g.*, effective user), the set of pending or blocked UNIX signals, and the current scheduling priority and policy.
- Application state, such as whether or not the current thread holds certain locks.

One of the most useful values to capture in conjunction with other values is the return address of the current procedure. This allows the value sampling system to identify values that are semi-invariant by call site.

To obtain the return addresses we take the simple approach of logging two values: the value in the return address register, and the value at the top of the stack. Because of the conventions followed by compilers that generate Alpha code, the return address is almost always to be found in one of these two places. Downstream analysis tools can deduce which, if either, of the two values is valid using the stack unwinding information present in the object file.

2.4 Customized Value Profiling

Our system can be customized in various ways. In particular, a user may specify what information to capture, how to transform it into value samples that are merged into the profile database, and how to format values for reporting.

To do this, the user writes a dynamically loadable *customization module*, which is loaded by DCPI's user-mode daemon. Via this "plug-in" module, users may specify what should be captured by the interpreter for each instruction opcode. One option is to capture nothing for particular opcodes, but usually some basic information is collected, including the PC and the 32-bit instruction code. In addition, users may opt to record one or more of the following: content of an explicitly named register, the instruction's operand or result, a memory operand's virtual address, and the latency of loads. For each instruction, the captured values form a *value tuple*. Thus, each time the interpreter runs, it generates a *tuple sequence* for the interpreted instruction sequence.

Tuple sequences generated in the interrupt handler are later processed by the user-mode daemon. For each sequence, the daemon calls a routine in the customization module to transform it into PC-value pairs that are merged into the profile database after data reduction. This transformation can be arbitrarily complex. For example, the daemon may transform value tuples consisting of the PC and operand address of load and store instructions into PC-value pairs (p, v) where v is the PC of another instruction accessing the same address as the instruction at p. (This is the idea behind the application in Section 3.2.) Our current implementation maintains only one value hotlist for each PC. It may be extended to maintain hotlists for different kinds of values (*e.g.*, data address and latency of a load) or for composite values (*e.g.*, address-latency pairs).

We modified the analysis tool *dcpilist* to report the most frequent values associated with each instruction in a format specified by the customization module. For example, the operands of floating-point instructions can be printed as floating-point numbers, rather than the default hexadecimal format.

We have written several customization modules, such as a module for capturing load latencies, as discussed in Section 3.3.

3 Experience

We have not used feedback from the value sampling system to direct automatic optimizations performed by the compiler. Nevertheless, we do have experience using it to highlight performance problems that programmers might then be able to address. Below we discuss some uses we expected, and some we did not.

3.1 Expected Uses

When collecting register values, we expected our system to provide information similar to that obtained from previous value profiling systems. We had no reason to believe that the quality of the data would be significantly better or worse than that obtained from those systems, though we might claim that our system is easier to use. We repeated the experiments of others only to verify that our value profiles agreed with prior work. We also looked at other programs to demonstrate that our profiles did give useful hints to programmers bent on optimization. We give only a few brief examples below.

Leveraging the ability of DCPI to pinpoint performance bottlenecks, our tools direct the programmer to places that both consume a significant amount of time, and which contain semi-invariant values. We showed that these tools made it straightforward for a programmer to discover specialization opportunities, such as those found by Calder *et al.* [4] in `mk88sim`.

We also found opportunities for specialization in the ray tracer `povray` when working on particular test images. An exponentiation routine was already specialized for a few integer exponents, but not the most common one. Adding an extra case yielded a 20% overall speedup. Similarly, specializing the routine `build-sturm` for degree 4 polynomials yielded a large improvement.

A smaller optimization opportunity was found in `gzip`, where a 2% speedup was achieved by noticing that a constant was being read repeatedly from a global variable.

3.2 Identifying Replay Traps

The Alpha 21264 processor attempts to execute memory access instructions as soon as possible, even if that means executing them out of order (that is, in an order other than program order). Part-way through the processor pipeline, a load or store may exceed some resource limit, or an architectural constraint on instruction ordering may be encountered which prevents the immediate issue of the instruction. In this case, the 21264 performs a *replay trap*, which aborts the instruction and all instructions that follow it in program order, and replays them from the fetch stage of the pipeline. Further details about replay traps can be found in the 21264 reference manual [1].

Replay traps can be quite expensive, and a programmer might care to know whether such an event is occurring in his inner loop. We have observed a few unusual programs in which the chip spends over half its time recovering from them. More commonly, one might expect to improve performance by a few percent by having good information about the causes of replay traps.

Some replay traps were of particular interest to us, because the chip's *ProfileMe* performance counters [7] do not provide all the information that one would wish for. The interesting replay trap types are:

- **Order:** When a load issues out-of-order before a store that accesses the same bytes, the load must be replayed to ensure that it fetches the stored bytes.

- **Size:** When a load follows a narrower store that accesses some of the same bytes, the load is replayed until the store has been merged with the other bytes.

- **Synonym:** If two off-chip memory accesses use addresses with the same cache index (*e.g.*, are congruent modulo 32K), one is replayed to avoid displacing data in the cache needed by the other.

In all these cases, a pair of memory accesses is involved. Even given one instruction of the pair, it can be difficult to identify the other simply by looking at the program text. For example, we have encountered an inner loop where a synonym trap was caused by a load from a global variable interacting with a load from a stack location.

Starting with the value sampling interpreter, we built a mechanism called *vreplay* to assist in these cases. The chip's ProfileMe hardware identifies the PC of one of the pair of memory accesses as one that incurred a large number of replay traps. The vreplay code identifies the likely PC of the other memory access, and which type of replay trap was involved.

The vreplay mechanism works by interpreting runs of instructions to detect accesses that may *potentially* conflict. Interpretation runs need to be long enough to include both instructions in a pair that cause a replay trap. On the 21264, the distance is bounded by the maximum number of instructions in flight (80), except for traps involving two loads; loads can retire before the data is back from memory.

Without expensive simulation, there is no way for the interpreter to know whether a replay trap would have really happened. However, combining data from the interpreter with data from ProfileMe samples ensures that the user's attention is directed only to instruction pairs that are in fact causing replay traps.

To eliminate some false alarms where accesses potentially conflict, the interpreter also tracks data dependencies between interpreted instructions. If there is a data dependence between two instructions that access memory, there can be no replay trap.

We found that we could not always interpret as long a run of instructions as we would have wished. On Tru64 UNIX, the TLB shootdown mechanism uses interprocessor interrupts (IPIs) to remove a TLB entry from each TLB in a multiprocessor. If a processor does not respond to the IPI within a time bound, the operating system crashes. This bound imposed a limit on the number of instructions that our uninterruptible, kernel-mode interpreter could process, and provided additional motivation for our user-mode value interpreter.

3.3 Load Latency Measurements

The value sampling system can measure the latencies of loads by reading a cycle counter before and after each one. Often, more than sixteen different latencies are observed for each load. To simplify the report generated, the user typically assigns latencies to bins using the known times for cache hits in various parts of the memory hierarchy. We use automatic programs to determine such interesting thresholds experimentally.

```
retdelay   PC instruction          vtot thld nv latencies
  0.0223 0x64 ldt  $f17, 8(t6)   23278  1.0  3 (94.26% D)(3.58% M)(2.15% B)
  ...
  0.0245 0x78 ldt  $f11, 0(t2)   14559  1.0  3 (84.91% M)(15.07% D)(0.01% B)
  ...
102.2877 0x94 mult $f11,$f17,$f17   0  0.0  0
```

Figure 1: Example of Load Latency value profile

Figure 1 is an example of a load latency value profile from a floating point benchmark. The *vtot* column is the number of value samples for the instruction; the *thld* column is the probability of each value sample being added to the hotlist; the *nv* column is the length of the hotlist; and the *latencies* column is the hotlist of binned load latencies, where "D" means a primary cache hit, "B" means a secondary (board-cache) hit, and "M" means a memory reference. The *retdelay* column comes from ProfileMe data and indicates the average number of cycles that the instruction stalled the CPU.

The `mult` instruction is an obvious bottleneck and is suffering from a cache miss that consumes more than 15% of all cycles in the benchmark. Because both operands (`f11` and `f17`) of the `mult` are the first uses of loads, the load latency profiles are essential to tell if the first, second, or both loads are missing. From the latency profile, it is clear that the first load usually hits and the second load usually goes out to memory.

A concern when recording load latencies is that our system might disturb the measurements so much as to make them worthless. We measured how much our system perturbs the primary data cache by creating a program that repeatedly touches each block in the cache, where a block is a <cache line, cache set> coordinate. The program uses a separate load instruction for each block. In an ideal world without interrupts, none of the these loads would get cache misses. The results are shown in Table 1.

Miss rate	Fraction of cache blocks
0%	88%
1-20%	4%
21-40%	2%
41-60%	2%
61-80%	1%
81-100%	3%

Table 1: Fraction of primary cache blocks experiencing various miss rates due to perturbation by value profiling.

The key point to notice in Table 1 is that about 90% of the cache blocks are never evicted by the value profiling system, while a small number are almost always evicted. The fraction of blocks evicted from the second level cache is lower due to its larger size. Thus, most load-latency values will be correct, while a few will be always be wrong. We expect these results could be improved by carefully tuning our interpreter to minimize the number of cache blocks touched.

3.4 Overhead Measurements

To assess the cost of value profiling, we measured how much it slowed down the CPU2000 benchmark suite on a 500 MHz Alpha 21264 machine. DCPI interrupts were generated on the average every 62K instructions. Table 2(a) shows the average (across the 12 integer benchmarks in the suite) slowdown in various cases.

Without value profiling (*no vprof*), the overhead is less than 4%. With basic value profiling (*vprof*), the interrupt handler interprets 4 instructions in one out of every two interrupts. The slowdown is nontrivial at about 10% but still much lower than that of instrumentation. This slowdown includes the effect of all DCPI-related

DCPI option	Average Slowdown %
no vprof	3.9
vprof	10.7
vprof with context	12.2
vreplay	5.9
vreplay with context	6.8

(a) Overhead of various profiling options

Interpret length	Interpret frequency	Average slowdown %
4	1/2	10.7
8	1/2	14.8
16	1/2	22.7
32	1/2	40.4

(b) Overhead of increasing interpret length

Interpret length	Interpret frequency	Average slowdown %
4	1/2	10.7
8	1/4	10.8
16	1/8	9.7
32	1/16	9.3
64	1/32	9.2

(c) Overhead for same average number of interpreted instructions per interrupt (2)

Table 2: Slowdown of CPU2000 integer benchmarks (in percent)

work: value and traditional profiling, driver and daemon processing. Although *vreplay* requires more complex processing than *vprof*, it costs less for two reasons. First, the handler interprets 128 instructions at a time (versus 4 for *vprof*) but compensates for that high cost by doing it only once every 128 interrupts. For most instructions, the driver emits no value samples to the daemon because there are no conflicting instructions to report, while in *vprof* it produces one sample for every interpreted instruction. For both *vprof* and *vreplay*, recording the return address information discussed in Section 2.3 (the "context") imposes a small extra cost as expected.

Tables 2(b) and (c) illustrate how we can manage the overhead by balancing how often to run the interpreter (*interpret frequency*, indicated as once every *n* interrupts) and how many instructions to interpret each time (*interpret length*). Table 2(b) shows the slowdown for different interpret lengths when the interpreter runs half the time. The slowdown increases approximately at the rate of 1% per instruction interpreted. Table 2(c) shows the slowdown for different cases that all lead to the same average number of interpreted instructions per interrupt. The overhead is roughly the same in each case but declines slightly if the interpreter is run less often because the per-interrupt cost is amortized over more instructions. Thus, we can increase the interpret length *and* keep overhead acceptable by interpreting less often. This is important because, in order to study interaction between instructions (as in vreplay), we may need to interpret a relatively long instruction sequence before getting any useful data at all.

4 Related work

Our value sampling work was primarily influenced by prior research on hardware mechanisms for value prediction and software techniques for value profiling. We were also motivated by growing interest in static and dynamic optimizers capable of exploiting value profiles.

Lipasti and Shen first introduced the idea of *value prediction* [13], proposing hardware that attempts to predict the next result value computed by an instruction based on a cache of previous result values for the same instruction. Their studies revealed a surprising amount of temporal locality; nearly half of all instructions produced the same result value computed during their last execution. Several subsequent proposals have been made for improved hardware value predictors [11, 15, 14].

Gabbay and Mendelson explored the use of profiling techniques to identify instructions which exhibit a high degree of value locality [10]. They showed that hardware value misprediction rates could be reduced by tagging the opcodes of predictable instructions, marking them as candidates for hardware prediction. Our low-overhead value sampling techniques could be used to provide even more detailed information to such hardware predictors.

Calder, Feller, and Eustace were the first to investigate software-based techniques for *value profiling* [4, 5, 9]. They used the ATOM [16] binary-rewriting tool to instrument each executable to be profiled, adding code to keep track of the most frequently occurring values computed by each instruction. A table of the top N values was maintained for each instruction, limiting storage requirements. A heuristic replacement policy was used to maintain the top N values approximately. When the table was full, the least frequently encountered value was evicted; half of the table was also periodically cleared to avoid pathological behavior with certain value sequences. In contrast to this *ad hoc* approach, which required tuning for good results, our application of the Gibbons and Matias sampling algorithms [12] provides a sound statistical basis for maintaining such value table hotlists. Instrumentation-based approaches also impose substantial overhead on profiled programs; Calder *et al.* reported average slowdowns ranging from a factor of 3.8 to a factor of 33, depending on various parameters. Our sampling-based approach imposes dramatically less overhead, enabling transparent value profiling on production systems.

Deaver, Gorton, and Rubin explored the use of limited value profile information for dynamic runtime code specialization [8]. Their *Wiggins/Redstone* optimizer identified hot spots using DCPI-based statistical profiling, and dynamically added instrumentation to frequently executed code to collect path and value information. Suitable traces were dynamically specialized and optimized as the program executed. Our user-mode value-sampling interpreter is an ideal match for such an optimizer.

Another approach aimed at transparent dynamic optimization was developed by Bala, Duesterwald, and Banerjia for their *Dynamo* system [3]. Instead of instrumentation, Dynamo relies on interpretation to observe program behavior without requiring modifications. As it interprets, Dynamo increments counters to identify hot instruction traces. Hot traces are selected for dynamic recompilation, which emits optimized code into a fragment cache. When the interpreter encounters a branch, it jumps to optimized native code in the fragment cache when it contains an entry for the branch target. Dynamo resumes interpretation when program execution leaves the fragment cache. Dynamo's use of limited interpretation has much in common with our own value sampling approach, although Dynamo does not collect value information, and its interpreter is not triggered by periodic interrupts.

There are many examples of systems that employ techniques that are essentially limited forms of value profiling. For example, run-time systems for languages such as Self [6] examine the types in use at call sites in order to replace indirect procedure calls with direct procedure calls and to pick subroutines specialized to those types.

5 Future Work

Many profile-driven optimizations could exploit value profiles. The usual example is specializing code sequences for frequently occurring values; another example is speculatively reducing the critical path of a high-latency computation by assuming it computes the most common values and then checking the assumption. On the Alpha, a simple but effective optimization would be to set the hint bits used to predict the target of an indirect jump based on the most common jump target.

The new types of values that our system can collect enable additional optimizations. Load latency value profiles could guide prefetching. The *vreplay* profiles, together with ProfileMe profiles, could be used to eliminate replay traps.

Our upcall handler could allow profile-driven optimizations to be done as the program is running, following the work of Deaver *et al.* [8]. Fixing jump hint bits and eliminating size and order replay traps are likely candidates because the required code analysis is local. Such optimizations are even more practical on a multiprocessor, where the optimization cost is amortized over many CPUs.

We see some bias in the distribution of interrupted PC locations despite the randomization of the interrupt period. This occurs because on modern processors the probability of an interrupt being delivered at a given PC depends not just on how often the instruction at that PC is executed, but also on other microarchitectural issues such as how often it causes a pipeline trap. Because our interpretation runs begin at the interrupted PC location, the distribution of value samples inherits this bias. For example, the loads in Figure 1 have significantly different numbers of value samples, despite being from the same basic block. We believe that we could eliminate this bias by interpreting a random number of instructions before sampling values.

6 Conclusion

We have presented a promising system for value sampling. We believe that our system makes it more convenient to collect value profiles than previous approaches. We have also experimented with new types of values that can be collected. In the remainder of this section we discuss what we felt went well or badly in our design.

The interpreter was a success. Our fears that it might be difficult to make it sufficiently reliable proved groundless. This contrasts with the "bounce-back" technique that we used before we introduced the interpreter. Although "bounce-back" involved much less code than the interpreter, it was tied to a particular hardware type and harder to implement correctly.

Some issues remain with the interpreter. The main one is that, when using the interpreter at interrupt level to diagnose replay traps, long interpretation runs can cause the operating system to crash, as described in Section 3.2. The use of user-space upcalls may be the answer to this problem. A minor irritation is that there are a few things that we cannot interpret, such as instructions that cause operating system traps, and instructions that modify the kernel stack pointer.

Gibbons and Matias' algorithm for maintaining hotlists simplified things. Employing a well-founded algorithm saved time that we might otherwise have spent in tuning and experimenting with more *ad hoc* approaches.

Our use of user-level upcalls for profiling shows promise, but we need more experience with it. We have already found that upcalls interact in interesting ways with UNIX signal handlers and exceptions. At present, we see no insurmountable problems.

7 Acknowledgements

We would like to thank Tim Mann for his comments and corrections on a draft of this report.

References

[1] "Alpha 21264 Microprocessor Hardware Reference Manual". Compaq Computer Corporation, June 1999. Order number EC-RJRZA-TE URL: http://gatekeeper.dec.com/pub/DEC/DECinfo/semiconductor/literature/21264hrm.pdf.

[2] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, D. Sites, M. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. "Continuous profiling: where have all the cycles gone?". In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 1–14, Saint-Malo, France, Oct. 1997. Also appears in *ACM Transactions on Computer Systems*, November 1997. URL: http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-016.html.

[3] V. Bala, E. Duesterwald, and S. Banerjia. "Transparent dynamic optimization". Technical Report HPL-1999-77, HP Laboratories, Cambridge, June 1999. URL: http://www.hpl.hp.com/techreports/1999/HPL-1999-77.pdf.

[4] B. Calder, P. Feller, and A. Eustace. "Value Profiling". In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 259–269, Dec. 1997. URL: http://www-cse.ucsd.edu/users/calder/abstracts/MICRO-97-VP.html or http://www.acm.org/pubs/articles/proceedings/micro/266800/p259-calder/p259-calder.pdf.

[5] B. Calder, P. Feller, and A. Eustace. "Value profiling and optimization". *Journal of Instruction Level Parallelism*, 1, Mar. 1999. URL: http://www.jilp.org/vol1/index.html or http://www-cse.ucsd.edu/users/calder/abstracts/JILP-99-VP.html.

[6] C. Chambers and D. Ungar. "Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language". In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989. *SIGPLAN Notices* 24(7), July 1989.

[7] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors". In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 292–302, Dec. 1997. URL: http://www.research.digital.com/SRC/dcpi/papers/micro30.ps or http://www.acm.org/pubs/articles/proceedings/micro/266800/p292-dean/p292-dean.pdf.

[8] D. Deaver, R. Gorton, and N. Rubin. "Wiggins/Redstone: An on-line program specializer". In *Proceedings of the IEEE Hot Chips XI Conference*, Aug. 1999. URL: http://rob.acol.com/˜wlug/files/deaver.pdf.gz.

[9] P. T. Feller. "Value profiling for instructions and memory locations". Master's thesis, University of California, San Diego, Apr. 1998. UCSD technical report CS98-581. URL: http://www-cse.ucsd.edu/users/calder/papers/PeterFellerThesis.ps.Z.

[10] F. Gabbay and A. Mendelson. "Can program profiling support value prediction?". In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 270–280, Dec. 1997. URL: http://www-ee.technion.ac.il/˜fredg/micro30.ps.gz or http://www.acm.org/pubs/articles/proceedings/micro/266800/p270-gabbay/p270-gabbay.pdf.

[11] F. Gabby and A. Mendelson. "Speculative execution based on value prediction". Technical Report TR-1080, EE department, Technion - Israel Institute of Technology, Nov. 1996. URL: http://www-ee.technion.ac.il/˜fredg/proptr.ps.gz.

[12] P. B. Gibbons and Y. Matias. "New sampling-based summary statistics for improving approximate query answers". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, Seattle, Washington, USA, June 1999. URL: http://www.bell-labs.com/user/matias/papers/sigmod98a.ps.

[13] M. H. Lipasti and J. P. Shen. "Exceeding the dataflow limit via value prediction". In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, Dec. 1996. URL: http://www.acm.org/pubs/articles/proceedings/micro/243846/p226-lipasti/p226-lipasti.pdf.

[14] G. Reinman and B. Calder. "Predictive techniques for aggressive load speculation". In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 127–137, Dec. 1998. URL: http://www.acm.org/pubs/articles/proceedings/micro/290940/p127-reinman/p127-reinman.pdf.

[15] Y. Sazeides and J. Smith. "The predictability of data values". In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 248–258, Dec. 1997. URL: http://www.acm.org/pubs/articles/proceedings/micro/266800/p248-sazeides/p248-sazeides.pdf.

[16] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools". In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994. URL: http://www.research.digital.com/wrl/techreports/abstracts/94.2.html.