Towards the Generalization of Value Profiling for High-Performance Application Optimization

Value Profiling made easy: MAQAO VPROF

Sylvain Henry Hugo Bolloré Emmanuel Oseret

Exascale Computing Research Laboratory
Campus Teratec
91680 Bruyères-le-Chatel, France
first.name@exascale-computing.eu

Abstract

Value profiling is a useful profiling method but it is often neglected because of the lack of methods and tools to perform it easily, efficiently and independently of the compilation chain. This is a pity because it can really increase program performance by analyzing their behavior at runtime and finding invariant or semi-invariant values that can lead to several optimizations (vectorization, specialization, memoization, etc.). We propose a method that relies on binary program patching to be generically applicable in addition to be easy to use, non-invasive and relatively fast. This work shows that value profiling can be generalized and that it should be integrated into the optimization process of high-performance applications.

1. Introduction

Suppose you want to optimize a scientific application (i.e. for which bottlenecks are in computational kernels, not in I/O handling). The usual steps to do that are:

1. Identify the "hot spots" – functions and loops where most of the time is spent and try to characterize the performance issue (compute bound, memory bound, etc.)
2. Optimize their source code as much as possible
3. Analyze their binary code to find new optimization opportunities that are specific to the targeted architecture or not trivial (e.g. best loop unrolling factors, vectorization hints, cache issues, etc.)
4. Try to find invariant or semi-invariant variables for a representative dataset (also called *value profling*) and optimize the code for these values: specialization, memoization, loop unrolling, etc.

To the best of our knowledge, the last step is often overlooked because of the lack of generic methods and tools to perform it easily and efficiently on production applications. With this paper, we would like to remedy this by presenting a novel approach to perform value profiling:

- We present an approach to perform value profiling based on binary patching: our solution takes production binary programs as input and patch them. Hence it does not require any modification of the source code nor any compilation and is very easy to use (Section 2). This approach allows us to profile applications with only a minimal impact while solutions that require modifications in the source code may lead the compiler to produce very altered binaries (e.g. different register allocation, inlining, etc.). Moreover, our solution is language agnostic and has been tested successfully with C, C++ and Fortran applications.
- We show that this approach can be applied to profile loops. In particular we give details on how to use it to retrieve the number of iterations and the number of cycles spent in a loop. We present an optimization that improves the accuracy of the measure of cycles spent in a loop in some cases and that improves over value profiling as performed in some production compilers (Section 3.3).
- We show that it is possible to know exactly how many times a path between two binary basic blocks is taken and that this information can be used to simplify static control-flow analysis or to optimize code: we can trim off paths that are seldom or never taken in the control-flow graph (Section 3.4).
- We show how to profile function calls (Section 3.5). This is particularly useful for function calls that are heavily used and that are quite expensive (e.g. calls to mathematical functions such as `exp`, `cos`, `sin`) We show how to retrieve call parameters and return values in a generic way (i.e. for almost any function, not only mathematical ones) and we show how these values can be used to implement specialization and/or memoization optimizations.
- We provide a tool that implements the proposed methods. It allowed us to apply them to production applications. A lot of effort have been put in it to ensure support for multi-threaded and distributed applications (OpenMP, MPI, etc.) and to design an interface as simple to use as possible.

This work is part of the effort to build a coherent tool suite called MAQAO which already provides: binary patching with MADRAS [11, 12]; hot spot identification with LPROF[1]; static

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] This tool will be fully described in an upcoming paper. Basically, it uses several ways to detect hot spots in applications: hardware performance counters, different sampling methods with or without binary patching, etc.

	Iterations	Cycles	Cycles/iteration
Total	94778586	3940196418	N/A
Min	3	40	13.33
Max	3	67168	22389.33
Average	3.00	124.72	41.57

Table 1. Simple statistics for a loop

binary analysis with CQA [4]; loop performance issue characterization with DECAN [7]; value profiling with VPROF (this paper).

2. Overview

To set the scene for this paper, in this section we give motivating examples of issues that can be encountered during the optimization of applications – we actually faced them and they drove the development of the value profiling methods described in the following sections of this paper. We briefly present the interface to use our tool to show that the presented method can be made easily accessible through a simple interface. The interested reader, however, is encouraged to read the user manual of our tool for a more comprehensive description of the available options not relevant to this paper and that have been left out.

2.1 Profiling loops

Suppose we have profiled an application (for instance in our case with MAQAO LPROF) and detected that a lot of time is spent in a given loop with the identifier 4155. We can use our tool to characterize it a little bit more:

```
maqao vprof ./prog loop-id=4155
```

This produces and executes a patched binary into which the loop identified with the identifier 4155 is profiled (instrumentation details are discussed in Section 3.3). The measured values are computed and stored in different ways, depending on the selected options, and finally written in a file which is used to produce reports.

For instance, with the previous command our tool reports that we entered the loop 31592862 times. It is what we call the number of instances. Simple statistics about the loop are reported too as shown in Table 1. We can already see that the number of iterations per instance is very low and constant (equal to 3). This can be used as a hint for the compiler to unroll, vectorize or specialize the loop.

The number of cycles varies a lot and our tool also reports a distribution of the instances per their average iteration cost (average number of cycles per iteration) as shown in Figure 1. Instances are clustered into bins with logarithmic bounds and a few instance numbers (not represented) are reported for each bin. These instance numbers can be used by other tools (e.g. MAQAO DECAN) to characterize some instances more precisely. For instance to detect the origin of the additional cost of the most cycle expensive instances (e.g. cache issues, etc.).

Our tool can also report more detailed statistics:

```
maqao vprof ./prog loop-id=4155 \
      --report-mode=full-stats
```

The instrumentation, however, is more expensive because we need to store on disk the number of iterations and the number of cycles for all loop instances. Each stored number uses 8 bytes of memory, hence for the given example with 31592862 instances for which we store both cycles and iteration count, the required disk space is 482MB. Results are shown in Table 2. Q1 and Q3 respectively represent the first and the third quartiles; D1 and D9 respectively represent the first and the ninth deciles.

	Iterations	Cycles	Cycles/iteration
Std dev	0.00	121.61	40.54
Median	3.00	67.00	22.33
Q1	3	59	19.67
Q3	3	97	32.33
D1	3	59	19.67
D9	3	303	101.00

Table 2. Full statistics for a loop

```
          values  Freq  FreqPercent
 [1,]  -0.120782  7346   18.2282878
 [2,]     -24.25   124    0.3076923
 [3,]     -28.75   120    0.2977667
 [4,]    -26.125   120    0.2977667
 [5,]     -25.75   120    0.2977667
 [6,]    -33.625   112    0.2779156
 [7,]     -29.25   112    0.2779156
 [8,]    -27.625   104    0.2580645
 [9,]    -23.125   104    0.2580645
[10,]     -18.25   100    0.2481390
[11,]     -40.75    96    0.2382134
[12,]     -39.25    96    0.2382134
[13,]     -33.75    96    0.2382134
[14,]     -33.25    96    0.2382134
[15,]     -31.75    96    0.2382134
[16,]    -23.625    96    0.2382134
[17,]    -20.125    96    0.2382134
[18,]     -36.25    88    0.2183623
[19,]    -32.625    88    0.2183623
[20,]    -30.625    88    0.2183623
```

Table 3. First 20 values for `exp` parameter sorted by decreasing frequency

Some loops have complex control flow and are hard to analyze statically and hard to optimize. To help reduce the complexity, we can measure how many times an edge between two blocks is traversed during the execution of the program:

```
maqao vprof ./prog loop-id=8775 \
      --enable-loop-paths
```

Figure 2 shows an example of the result on the second hottest loop of a production application which has an interesting control flow. We see that we can trim off a branch which is never taken with the tested dataset. We also know which basic blocks should be optimized in priority because most paths traverse them.

2.2 Profiling function calls

Our team had to optimize a production application whose profiling report showed that a lot of time was spent in calls to the libc exponential function (`exp`). We used our tool to profile the calling parameters:

```
maqao vprof ./prog \
      --calls="double exp(double x);"
```

Profiling revealed that there were 22722 calls to the function but only 403 different input values. Table 3 presents the first twenty most occurring parameters. We can see that `exp` is called with the same parameter more than 18% of the time. Moreover, the successive calls seem to follow a pattern as can be seen in Figure 3.

To optimize the application, we could:

- change the source code to avoid these redundant calls. In the end, the application was optimized with this method as it was a trivial change to make after it had been detected.

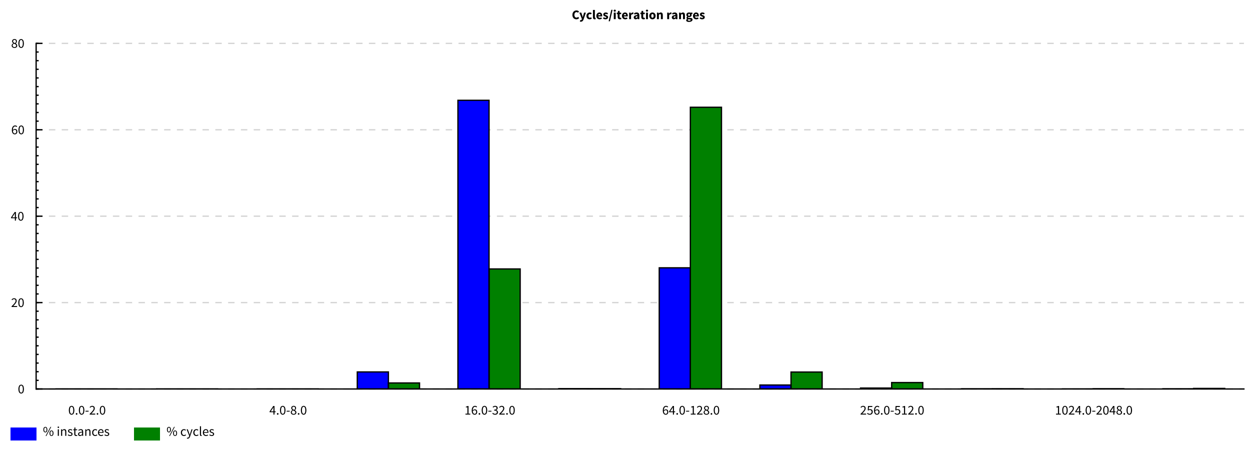

Figure 1. Instance distribution per average iteration cost (in cycles)

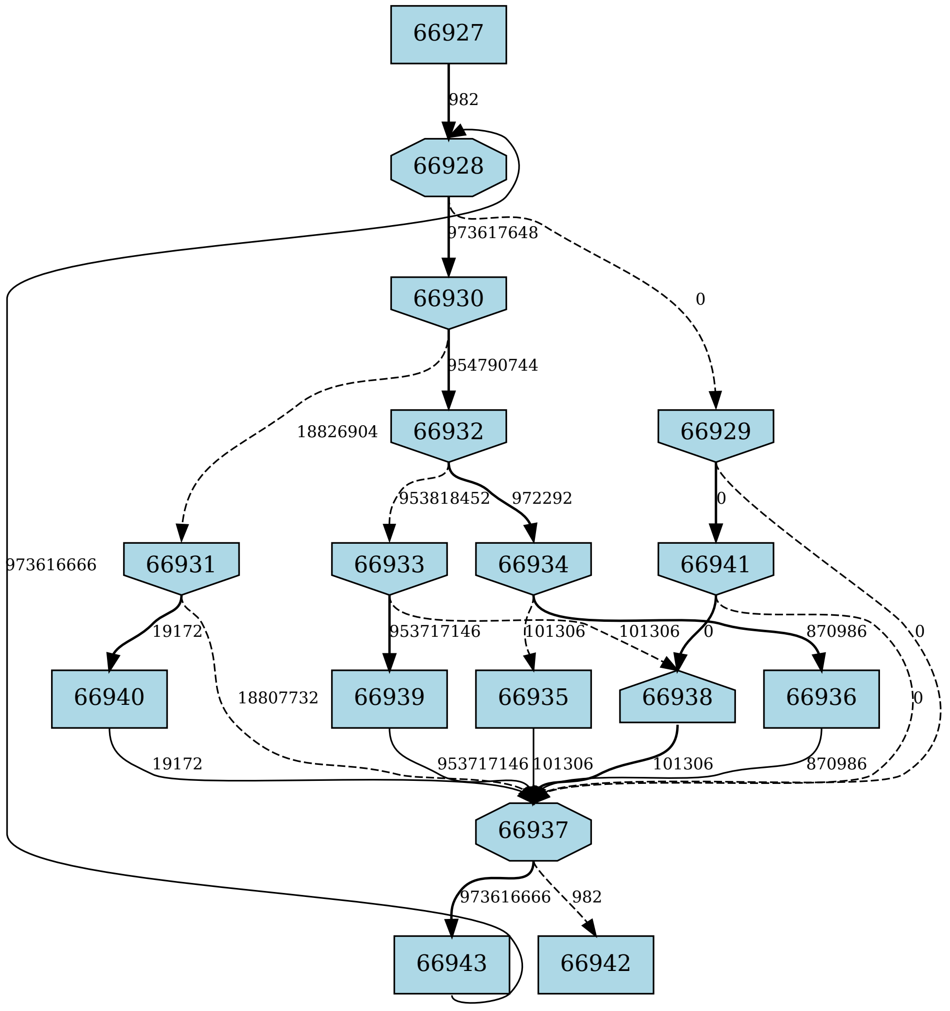

Figure 2. Loop paths example. The shape of each block depends on the number of predecessors and successors. Bold plain edges indicate natural flow (i.e. next instruction); light plain edges indicate unconditional jumps; dashed edges indicate conditional jumps. Numbers annotating each edge indicate the measured number of times each edge is traversed during the whole execution of the program.

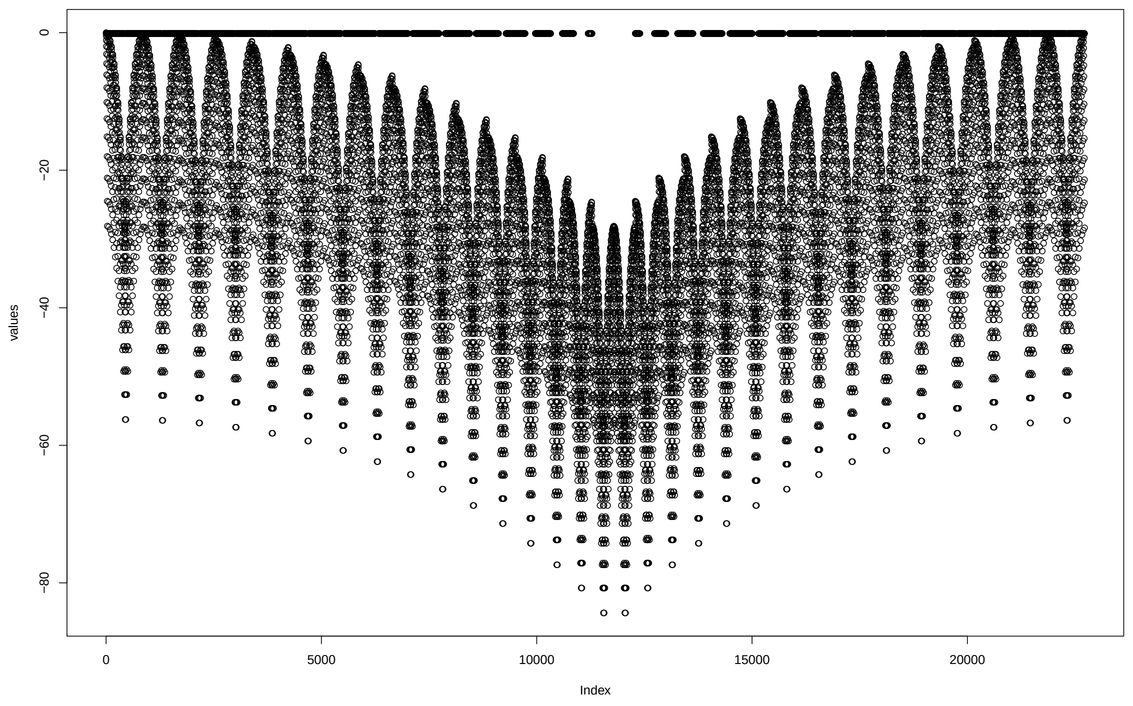

Figure 3. Values of the `exp` parameter for the successive 22722 calls. Only 403 different parameter values are used and we can observe a pattern. The almost straight "line" on top represents the most occurring parameter which is used for 18% of the calls (cf Table 3).

- specialize the call to `exp` in order to hardcode the most occurring value(s)
- use memoization for a certain number of parameter/result couples for the `exp` function

Memoization is useful to avoid hard coding some values into the code as it is the case with specialization. The idea is to have a cache of some previous parameter/result couples and to avoid calling a costly function if we can get the result from the cache (in this case we have a "cache hit"). The major advantage is that performance can be improved for different datasets if the call pattern stays the same even if calls are made with different values for each dataset.

To use memoization, we need to know how many parameter/result couples to store (in a hash table for instance). In Figure 4 we plot the distance (in number of calls to `exp`) between calls with the same parameter for the first twenty most occurring parameters:

- the dotted ranges indicate the full range of distances (e.g. for the second value, the distance between two successive identical calls ranges from about 15 to about 2000 calls).
- the box indicates the quartiles: the bottom of the box is the first quartile and the top of the box is the third quartile
- the horizontal line in the box represents the median

If we use a cache of the last 250 parameter/result couples for the `exp` function, we see that we will have a cache hit rate above 75% for 18 of the 20 most frequent values (cf the horizontal line in Figure 4).

Similarly, in the same program `log` was called 14691 times from two call sites with only 2 different values (one per call site). We used the binary patching method to replace calls to `log` and to `exp` always performed with the same parameter with a simple load of the correct value from memory to a register.

3. Instrumentation details

Patching binary programs is hard. Hence our tool relies on the MADRAS component [11, 12] of the MAQAO tool suite to do it. MADRAS can insert instructions into a program at almost arbitrary places and takes care of disassembling the original program, moving basic blocks, modifying relocation tables and assembling the final program.

MADRAS allows the insertion of new global variables and new thread-local storage (TLS) variables into programs (the latter has been developed in the context of this work to make our tool support multi-threaded applications). We use these features to add variables for the various data structures we need that contain counters, file handles, etc.

Describing exactly how MADRAS work is out of the scope of this paper. Interested readers are invited to read Cédric Valensi's thesis [11] or this older technical report [12]. A new paper presenting recent developments is expected soon.

3.1 Instruction insertion

The main issue with binary program instrumentation is to keep the original semantics of the program and to be as close as possible to the original performance, especially when we measure cycles if we want our measures to be meaningful.

To keep the original semantics of the program, the code we insert must not change the values of the registers and memory used by the program. One way to do that is to save the registers our code use before their use and to restore them afterwards. Now the issue is to find a location where to store registers.

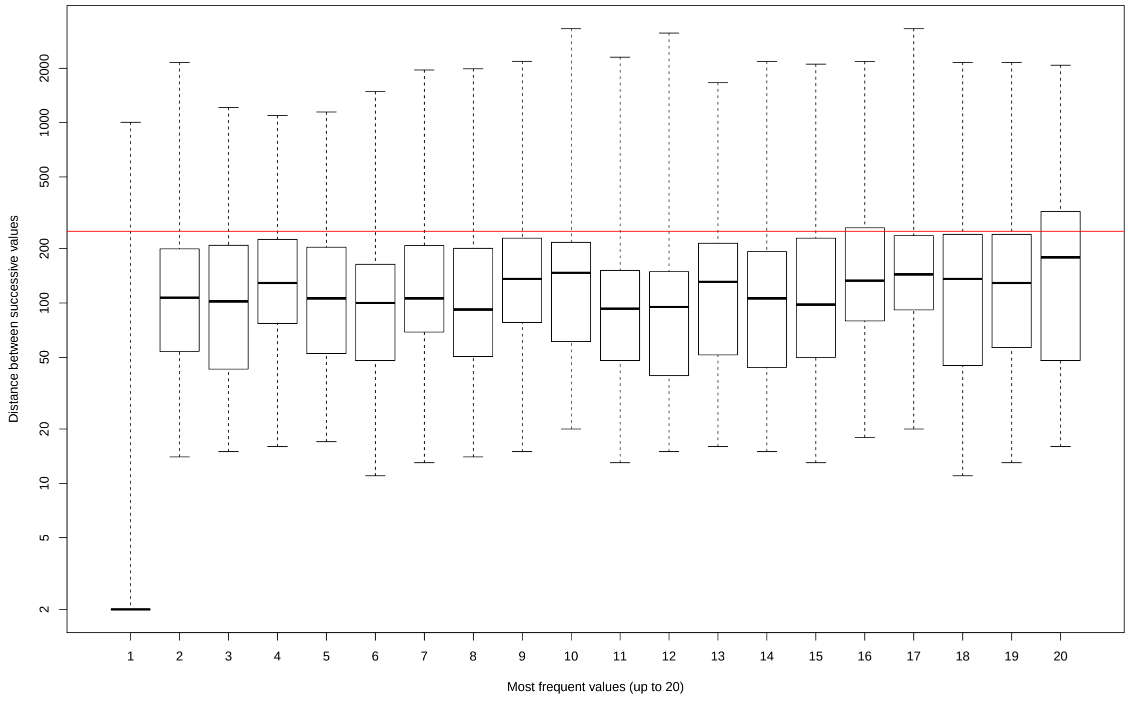

Figure 4. Periodicity of the most frequent values

Hopefully most languages used in high-performance computing (i.e. C, Fortran, C++) use the stack in the same manner: to store local variables and to pass parameters for function calls. If we move the stack pointer after the local variables of the function we are inserting code into (and after its *red zone* on x86-64 architectures [1]), we have an already allocated memory space that we can use without perturbing the program execution.

To insert some instructions in a program on the x84-64 architecture, we use the pattern shown in Figure 5 to store and restore the context (i.e. to avoid altering the program semantics). The code to align the stack before the call to `fxsave` and to store the old position has been left out for clarity. Some of the instructions used in this pattern are very costly and our objective is to remove those that are not strictly necessary.

A possible approach is to use a *live registers analysis* to detect registers that are alive at the address of the insertion: registers that are not alive can be used without being saved. To illustrate this, suppose we are inserting some code before the instruction `xor %rax, %rax`, then we can use the %rax register freely because it will be written before being read again in the following code.

Registers that are not used by the inserted instructions don't have to be stored either. This method is used to insert instructions to measure cycles in Section 3.2. Similarly if the inserted instructions don't modify flags or if the modified flags are not "live" (i.e. they are written before being read by the following instructions), then we can avoid saving and restoring them with the costly `pushfq` and `popfq` instructions. Finally, if we don't have to store any register nor the flags register, then we can avoid moving the stack pointer. This method is used to insert the iteration counter in Section 3.3.

3.2 Counting cycles

Measuring the number of cycles spent in a code on the x86-64 architecture is described in an Intel white-paper [9]. `rdtsc` and `rdtscp` instructions are used. These instructions return their result in the RAX and RDX registers and modify the flags. Figure 6 shows how they are inserted into a binary with their surrounding context saving instructions.

Our tool provides an option to use serializing instructions to avoid counting out-of-order instructions fetched before the start counter or fetched in advance after the stop counter. We use the `cpuid` instruction (cf Intel's white paper) and we save and restore RBX and RCX registers accordingly.

Our measure of the cycles includes cycles spent for instructions that are not in the original program but are used to save and restore the context and to store the result of `rdtsc`. As this cost is almost constant, our tool inserts some code to evaluate the minimal cost in cycles of these extraneous instructions at the beginning of the program and subtracts this value from the further measured values.

3.3 Counting iterations

Counting the number of iterations must be as cheaper as possible, especially when we measure the number of cycles spent in the loop at the same time: the reported time includes the cost of the iteration counter code. Our tool proposes a mode where the two measures are performed separately (i.e. the application is patched and executed twice). This approach, however, is limited because to be able to associate cycles measures and iterations measures, it requires that the application has a deterministic behavior. In particular, multi-thread and multi-process applications are not supported with this mode because the thread/process numbers are not deterministically assigned.

The first step to count the number of iterations of a loop is to identify the loop pattern. MAQAO uses a variant of the algorithm from [14] to identify loops at the binary level, hence it is not always clear what an iteration is: some loops have several entries, several

```
lea rsp, [rsp - 0x200]  ; skip red zone
pushfq                  ; push flags register
push ...                ; push registers
...
fxsave [rsp]            ; save vector registers

; inserted instructions

fxrstor [rsp]           ; restore vector registers
pop ...                 ; pop registers
...
popfq                   ; pop flags register
lea rsp, [rsp + 0x200]  ; restore original stack
                          pointer
```

Figure 5. Generic instruction insertion pattern on x86-64 architecture

Mode	cycles/iteration
Original loop (without iteration counter)	90
With iteration counter in memory	131
With iteration counter in a register	90

Table 4. Measured cycles/iteration to show the effect of the optimization against the 4K aliasing issue

exits and several back edges (i.e. edges that goes to a potentially already visited block in the control flow).

Hopefully, in most cases loops have a single entry block and a single exit block and we can easily identify where to insert the iteration counter incrementation code. In the worst case – which is also the least common – we cannot do that and we insert the code on every back edge.

On x86-64 architectures, the `inc` instruction can be used to increase the value in a register or at a memory address. This instruction modifies some flags hence the first optimization we implemented was to find an instruction of the original code that sets or leaves undefined the same flags. Moreover the instruction has to be executed in each iteration of the loop. If we find such an instruction, we add the `inc` instruction before it without saving any register nor the flags register.

The `inc` instruction performs three micro-operations: load, add, store. When the source/target is a memory address, as the address is always the same during the whole execution of the loop, the instruction is subject to a store forwarding stall issue (also known as 4K aliasing [5]). Basically, the memory accesses are serialized and the performance drops, leading to overly increased cycle measures.

Our tool uses an optimization to avoid this issue in some cases. It tries to detect a general purpose register that is not used in the loop and it uses it to store the iteration counter. If it cannot find any, it falls back to storing it in memory. Before the loop, the value in the register is saved and the register is set to 0; after the loop, the value in the register (the number of iterations) is saved and the original value of the register is restored. The is implemented with two calls to the `xchg` instruction with the selected register and the memory location for the counter as parameters. Figure 4 shows the results we obtained without and with this optimization on a specific case that was at the origin of this optimization. We can see that this optimization is very effective as it makes the cost of the iteration counter code disappear: the avoidance of the 4K aliasing issue keeps the original out-of-order execution behavior of the code.

```
lea rsp, [rsp - 0x200]
pushfq
push rax
push rdx

rdtsc
sal rdx, 0x20
or rax, rdx
mov [rip - 0x3d9], rax

pop rdx
pop rax
popfq
lea rsp, [rsp + 0x200]
```

Figure 6. Example of inserted instructions to measure cycles spent in a loop on x86-64 architecture (this is only the first insertion before the loop)

3.4 Tracing paths

Tracing control-flow paths in a loop or in any set of basic blocks is very similar to the instrumentation to count iterations. The difference is that there is a counter for each couple of successive blocks and that counters are never reseted. We insert counter incrementation code for each counter at appropriate locations.

Similarly to the iteration counter, if the flags register is not ”alive” at the beginning of the target basic block, we can avoid storing and restoring it.

The resulting file uses three 8-byte words per couple of blocks to store the two block identifiers and the measured value. It is then easy to trace annotated control-flow graphs as in Figure 2.

This method can be very expensive and must be used only on pathological cases. As an alternative, loops with too many blocks/instances/iterations that cannot reasonably use this method can use a statistical sampling approach as provided by MAQAO LPROF to determine loop hot paths.

3.5 Function call parameters

Function call instruction consists in inserting instructions before and after a `call` instruction targeting the function we are interested in. The location of call parameters and returned values is fully described in [1] for the x86-64 architecture. As long as the called function respects the calling convention described in the document, we know which registers or stack location to read.

By using the method to insert instructions described in Section 3.1 without any optimization to reduce the number of saved registers, we know the location of saved register values on the stack, we read the appropriate ones to retrieve the call parameters and the returned values and we store them in a file. Finally, with these files we can compute statistics and generate graphs as presented in Section 2.2.

To provide a simple interface, our tool parses the given function declarations in C (e.g. ”double log(double x)”), automatically infers where to read parameters (registers, stack, etc.) and uses the parameter names in the output reports. This makes it very convenient for users of the tool because they don't need to be aware of the calling conventions and it ensures that when our tool will be ported to other architectures, it will keep the same interface: only the application binary interface (ABI) used internally will change.

4. Related works

As far as we know, the term ”Value profiling” has been coined in 1997 by Brad Calder et al. [2]. Futher experiments performed on

Alpha architectures showed that up to 21% execution time speedup could be obtained by using value profiling and specialization [3]. These experiments used the generic ATOM framework [10] to perform binary instrumentation at link time on Alpha architectures.

Watterson et al. [13] built on this work in the context of the *alto* [8] link-time optimizer (still on Alpha architectures). The idea was to filter the results of the value profiler depending on their expected use in the following optimization passes in order to reduce the memory consumption and to speed-up the value profiling phase. Even if memory is a lot cheaper these days, this optimization is interesting and our tool combined with an optimizer could also use it. Our tool already provides several options to precisely select what has to be profiled and which methods to use so that cheap profiling can be applied broadly to reduce the exploration space while costly methods which require lot of memory (e.g. returning the number of iterations and the number of cycles for all instances) can be reserved to a restricted set of pathological cases.

Khan [6] showed that specialization guided by value profiling could lead to performance gains: he reports an average 3% gain with the SPEC 2000 benchmarks[2] on the Itanium-II architecture which was better than the results obtained with the Intel C Compiler at the time.

The GNU C Compiler (GCC) and compatible compilers such as Intel compilers support basic function call instrumentation with the `--instrument-function` parameters. The major drawback of the compiler approach is that it requires a recompilation of the program while our approach directly patches the binary program. Moreover, to the best of our knowledge the user cannot select the functions to instrument, nor dump the function parameters and returned values.

Intel Fortran compiler (`IFORT`) supports value profiling for loops. At the time of writing, however, the latest version (16.0) is still subject to the 4K aliasing issue presented in Section 3.3. We don't think there is any obstacle to implement the optimization we present in this paper in a future version of their compiler though.

5. Conclusion

We have presented a way to easily perform value profiling on programs. The focus has been put on loops and function calls because they are at the core of high-performance scientific programs but could be extended to work with other aspects (I/O, etc.).

The method we presented performs binary patching. Hence it is applicable to programs written in different languages, it does not require any modification of the compilation chain and it does not require programs to be recompiled explicitly for value profiling. Moreover, this technique ensures that we profile exactly the code that is generated by the compiler without the profiling probes.

We showed that we have been able to optimize the method to measure the loop iteration count to the point it becomes cycle-wise unobservable in some cases while even production compilers performing value profiling are still subject to highly penalizing store-forwarding issues in the same context.

We briefly presented the implementation and the user interface of a tool called VPROF which is integrated into the MAQAO tool suite[3]. It shows that the proposed method is made easy to use through a simple interface. This supports our claim that value profiling should be generalized as a diagnostic method because the potential optimization opportunities compared to the difficulty and cost (time, resources, etc.) to use the method is very favorable. We only described a few options of our tool relevant for this paper. Interested readers should refer to the user manual for a comprehensive description of the options.

[2] www.spec.org

[3] www.maqao.org

Our tool provides support for multi-threaded and distributed applications (OpenMP, MPI, etc.). It presented some challenges because the inserted profiling code had to be able to use thread-local storage (TLS) variables and to be able to produce reports from several potentially big trace files. A lot of efforts have been put in the development of our tool and the MAQAO tools it depends on to make this possible.

Some tools of the suite – such as DECAN, the loop instance performance issue characterizer, or CQA, the static binary code analyzer – already use VPROF to identify interesting loops/functions and to trim off unused control-flow paths during loop analysis. Hence our tool is already well integrated and has been tested in multiple settings.

5.1 Further work

Future research includes automatically using profiling information returned with our method. Profile guided optimization (PGO) is a well-known technique worth exploring at the compiler level. Another more original approach would be to use binary patching to improve performance. For instance by adding memoization and/or specialization to a program automatically. To the best of our knowledge, this would be the first time such optimization would be performed using an automatic language agnostic method.

The current implementation is limited to x86-64 architectures (with SSE and AVX vector extensions). We would like to experiment with some of the techniques presented here on other architectures with different instruction sets (e.g. ARM).

References

[1] System V Application Binary Interface 0.99, 2013. URL `www.x86-64.org`.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 259–269. IEEE, 1997.

[3] B. Calder, P. Feller, A. Eustace, et al. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1(1):1–6, 1999.

[4] A. S. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue. Cqa: A code quality analyzer tool at binary level. In *High Performance Computing (HiPC), 2014 21st International Conference on*, 2014.

[5] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers, 2014.

[6] M. A. Khan. Improving performance through deep value profiling and specialization with code transformation. *Computer Languages, Systems & Structures*, 37(4):193–203, 2011.

[7] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby. Quantifying performance bottleneck cost through differential analysis. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 263–272. ACM, 2013.

[8] R. Muth. *Alto: A platform for object code modification*. PhD thesis, UNIVERSITY OF ARIZONA, 1999.

[9] G. Paolini. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. Technical report.

[10] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[11] C. Valensi. *A generic approach to the definition of low-level components for multi-architecture binary analysis*. PhD thesis, 2014.

[12] C. Valensi and D. Barthou. Madras: Multi-architecture binary rewriting tool. Technical report.

[13] S. Watterson and S. Debray. Goal-directed value profiling. In *Compiler Construction*, pages 319–333. Springer, 2001.

[14] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *Static Analysis*, pages 170–183. Springer, 2007.