Estimating the Empirical Cost Function of Routines with Dynamic Workloads

Emilio Coppa
Sapienza University of Rome
coppa@di.uniroma1.it

Camil Demetrescu
Sapienza University of Rome
demetres@dis.uniroma1.it

Irene Finocchi
Sapienza University of Rome
finocchi@di.uniroma1.it

Romolo Marotta
Sapienza University of Rome
romolo.marotta@gmail.com

ABSTRACT

A crucial aspect in software development is understanding how an application's performance scales as a function of its input data. Estimating the empirical cost function of individual routines of a program can help developers predict the runtime on larger workloads and pinpoint asymptotic inefficiencies in the code. While this has been the target of extensive research in performance profiling, a major limitation of state-of-the-art approaches is that the input size is assumed to be determinable from the program's state prior to the invocation of the routine to be profiled, failing to characterize the scenario where routines *dynamically* receive input values during their activations. This results in missing workloads generated by kernel system calls (e.g., in response to I/O or network operations) or by other threads, which play a crucial role in modern concurrent and interactive applications. Measuring dynamic workloads poses several challenges, requiring shared-memory communication between threads to be efficiently traced. In this paper we present a new metric and an efficient algorithm for automatically estimating the size of the input of each routine activation. We provide examples showing that our metric allows the estimation of the empirical cost functions of complex applications more precisely than previous approaches. An extensive experimental investigation on a variety of benchmarks shows that our metric can be integrated in a Valgrind-based profiler incurring overheads comparable to other prominent heavyweight dynamic analysis tools.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Algorithms, Measurement, Performance.

CGO '14, February 15 - 19 2014, Orlando, FL, USA.
 ACM 978-1-4503-2670-4/14/02 $15.00.

Keywords

Asymptotic analysis, instrumentation, performance profiling, Valgrind, workload characterization.

1. INTRODUCTION

Performance profilers collect information on running applications and associate performance metrics to software locations such as routines, basic blocks, or calling contexts [1, 9, 18]. They play a crucial role towards software comprehension and tuning, letting developers identify hot spots and guide optimizations to portions of code that are responsible of excessive resource consumption.

Unfortunately, by reporting only the overall cost of portions of code, traditional profilers do not help programmers to predict how the performance of a program scales to larger inputs. To overcome this limitation, some recent works have addressed the problem of designing and implementing performance profilers that return, instead of a single number representing the cost of a portion of code, a function that relates the cost to the input size (see, e.g., [5, 8, 23]). This approach is inspired by traditional asymptotic analysis of algorithms, and makes it possible to analyze – and sometimes predict – the behavior of actual software implementations run on deployed systems and realistic workloads. Some of the proposed methods, such as [8], perform multiple runs with different and determinable input parameters, measure their cost, and fit the empirical observations to a model that predicts performance as a function of the workload size. More recent approaches made a step further, tackling the problem of automatically measuring the size of the input given to generic routines [5, 23], collecting data from multiple or even single program runs.

As observed in [5] and [23], a current limitation of profilers that estimate cost functions is that they ignore any communication between threads and data received via system calls from the OS kernel, failing to accurately characterize the behavior of routines executed in the context of modern concurrent and interactive applications. In this paper we show how to overcome this limitation.

Contributions. The main contributions of this paper can be summarized as follows:

- We propose a novel metric, called *dynamic read memory size*, that accurately and automatically estimates the size of the input of a routine activation, taking into account *dynamic workloads* produced by memory

stores performed by other threads and by the OS kernel (e.g., in response to I/O or network operations).

- We provide real case studies, based on a database management system (`MySQL`) and on an image processing tool (`vips`), showing that the new metric allows estimating the empirical cost function of complex applications more precisely than previous approaches.
- We present an efficient profiling algorithm that computes the dynamic read memory size of each routine activation, producing as output a set of performance points that relate the cost of each routine to the observed distinct input sizes.
- To prove the feasibility of our approach, we implemented our metric in `aprof` [5], a Valgrind open-source profiler.
- We performed an extensive set of experiments on a large variety of benchmark suites, including PARSEC and SPEC OMP2012, evaluating the benefits of our metric and showing that our tool can characterize the nature of dynamic workloads on the considered benchmarks, incurring overheads comparable to the other tools in the Valgrind suite.

2. DYNAMIC INPUT SIZE ESTIMATION

A crucial issue in automatically estimating the cost function of a routine is the ability to infer the size of the input data on which each activation operates. This can be done for workloads that are stored in memory prior to the routine activation to be profiled using the *read memory size* metric introduced in [5]:

DEFINITION 1. *Let r be a routine activation by thread t. The* read memory size $\text{RMS}_{r,t}$ *of r with respect to t is the number of distinct memory cells first accessed by r, or by any descendant of r in the call tree, with a read operation.*

The intuition behind this metric is the following. Consider the first time a memory location ℓ is accessed by a routine activation r: if this first access is a read operation, then ℓ contains an input value for r. Conversely, if ℓ is first written by r, then later read operations will not contribute to increase the RMS since the value stored in ℓ was produced by r itself. Estimates of the input size can be then used to automatically produce performance charts: e.g., for each distinct input size n of a routine r, we could plot the maximum time spent by an activation of r on input size n (worst-case cost plot).

The RMS fails to properly characterize the input size of routine activations under dynamic workloads. Consider, as an example, the concurrent execution described in Figure 1a: routine f in thread T_1 reads location x twice, but only the first read operation is a first access. Hence, $\text{RMS}_{f,t} = 1$. Routine g in thread T_2, however, overwrites the value stored in x before the second read by f: this read operation gets a value that is not produced by routine f itself and that should be therefore regarded as new input to f. The same drawbacks discussed in the example above arise when one or more memory locations are repeatedly loaded by a routine with values read from an external source, e.g., network or secondary storage. To overcome these issues, we propose

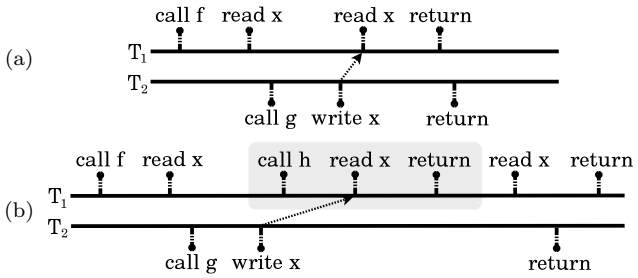

Figure 1: Dynamic read memory size examples.

a novel metric for estimating the input size, which we call *dynamic read memory size*.

DEFINITION 2. *Let r be a routine activation by thread t and let ℓ be a memory location. A read operation on ℓ is:*

- *a* first-read*, if ℓ has never been accessed before by r or by any of its descendants in the call tree of thread t;*
- *an* induced first-read*, if no previous access to ℓ has been made by t since the latest write to ℓ performed by a thread different from t, if any.*

DEFINITION 3. *Let r be a routine activation by thread t. The* dynamic read memory size $\text{DRMS}_{r,t}$ *of r with respect to t is the number of operations performed by r that are first-reads or induced first-reads.*

For the sake of presentation, in this section we are assuming that the OS kernel runs as a separate thread so that the DRMS includes also inputs fed to the application from external sources. This assumption will be relaxed in Section 3.2.

We notice that the RMS coincides with the number of operations that are first-reads and therefore

$$\text{DRMS}_{r,t} \geq \text{RMS}_{r,t} \tag{1}$$

for each routine activation r and thread t.

Example. Consider again the example in Figure 1a: we have $\text{DRMS}_{f,T_1} = 2$. The first read operation on x is indeed a first-read, while the second one is an induced first-read, as T_2 has modified x since the previous access to x by T_1.

Now consider Figure 1b. In this case $\text{RMS}_{h,T_1} = 1$ and $\text{RMS}_{f,T_1} = 1$: function f performs three read operations on x (one of which through its subroutine h), but only the first one is a first-read and contributes to its RMS. Conversely, it holds $\text{DRMS}_{f,T_1} = 2$. The read operation by h is indeed an induced first-read for f (similarly to the previous example), while the third read is not: between the latest write operation on x performed by thread $T_2 \neq T_1$ and the third `read`(x), f has already accessed x through its descendant h.

We also have $\text{DRMS}_{h,T_1} = 1$. Notice that the read operation in h could be regarded both as a first-read and as an induced first-read with respect to h.

We now describe two common software patterns whose entire workloads are dynamically generated.

Pattern 1: Producer-Consumer. Producer-consumer is a classical pattern in concurrent applications. The standard implementation based on semaphores (see, e.g. [19]) is shown in Figure 2, where producer and consumer run as different

```
procedure producer()
 1: while (1) do
 2:     wait(empty)
 3:     wait(mutex)
 4:     x = produceData()
 5:     signal(mutex)
 6:     signal(full)

procedure consumer()
 1: while (1) do
 2:     wait(full)
 3:     wait(mutex)
 4:     consumeData(x)
 5:     signal(mutex)
 6:     signal(empty)
```

Figure 2: Producer-consumer pattern: at iteration n, $\text{RMS}_{\texttt{consumer},t} = 1$ while $\text{DRMS}_{\texttt{consumer},t} = n$.

threads and routines `produceData` and `consumeData` write to and read from memory location x, respectively (the implementation can be easily extended to buffered read and write operations). For simplicity of exposition, we will not consider memory accesses due to semaphore operations. With this assumption, $\text{RMS}_{\texttt{consumer},t} = 1$, since the consumer repeatedly reads the same memory location x. Conversely, the dynamic read memory size gives a correct estimate of the consumer's input size: whenever `producer` has generated n values written to location x at different times, we have $\text{DRMS}_{\texttt{consumer},t} = n$. Indeed, all read operations on x are induced first-reads: thanks to the interleaving guaranteed by semaphores, each `read`(x) in `consumeData` is always preceded by a `write`(x) in `produceData`.

```
procedure streamReader()
 1: let b a buffer of size 2
 2: for i = 1 to n do
 3:     fill b with external data
 4:     consumeData(b[0])        // read and process b[0]
```

Figure 3: Buffered read from a data stream: after n iterations, $\text{RMS}_{\texttt{externalRead},t} = 1$ and $\text{DRMS}_{\texttt{streamReader},t} = n$.

Pattern 2: Data Streaming. The example in Figure 3 describes the case of buffered read operations from a data stream. Procedure `streamReader` loads $2n$ values (line 3): this is done by the operating system that fills in buffer b with fresh data at each iteration. Only one of the two values loaded at each iteration is then read and processed at line 4. Hence, at the end of the execution $\text{DRMS}_{\texttt{streamReader},t} = n$, due to the n induced first-reads at line 4. Conversely, $\text{RMS}_{\texttt{streamReader},t} = 1$: data items are loaded across iterations on the same two memory locations $b[0]$ and $b[1]$, but only $b[0]$ is repeatedly read.

2.1 Empirical Cost Function Estimation: Case Studies

In this section we discuss examples showing that the DRMS metric can greatly improve accuracy in characterizing the empirical cost functions of prominent real applications.

MySQL. As a first case study, we consider the MySQL DBMS. We performed a simple experiment with a query operation that selects all tuples in a table, repeating the query on tables of increasing sizes. At each query, processed by routine `mysql_select`, tuples are partitioned into groups. Each group is loaded into a buffer through a kernel system call and is then read by `mysql_select`. The RMS does not count repeated buffer read operations: hence, the input size on larger tables is exactly the same as in smaller ones (it roughly coincides with the buffer size), while the performance cost grows due to the larger number of buffer loads. Figure 4 shows the cost plots of routine `mysql_select` (measuread as number of executed basic blocks) resulting from both the DRMS and the RMS. The DRMS plot correctly characterizes the linear cost trend of `mysql_select`, while the RMS plot suggests a false superlinear trend.

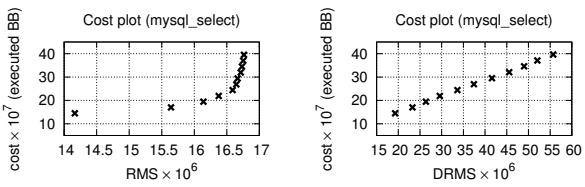

Figure 4: Function `mysql_select` of `MySQL`: worst-case cost plots respectively obtained using RMS or DRMS as an estimate for the input size.

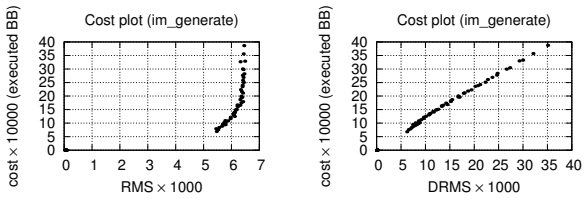

Figure 5: Function `im_generate` of `vips` (PARSEC 2.1): worst-case cost plots respectively obtained using RMS or DRMS as an estimate for the input size.

vips. Routine `im_generate` of benchmark `vips` of the PARSEC 2.1 suite [3] shows an analogous effect on the `simlarge` reference input (see Figure 5). In this case the induced first-reads not counted in the RMS are due to the interaction between threads via shared memory. In both examples, the RMS plot yields clues to an asymptotic bottleneck, which instead does not actually exist.

The ability to accurately estimate cost functions crucially depends on the number of distinct input size values collected for each routine: each value corresponds to a point in the cost plot generated by the profiler, and plots with a small number of points do not clearly expose the behavior of the routine. In our experiments, we observed that in many cases the DRMS yields a much larger number of distinct input size values than the RMS. An example is the `vips` routine `wbuffer_write_thread` shown in Figure 6: out of 110 distinct routine calls, the charts plot the maximum costs for all calls having the same RMS/DRMS value. As shown in Figure 6a, the RMS collapses all input sizes onto two distinct values, retaining the maximum cost over 65 calls on RMS 67 and the maximum cost over 45 calls on RMS 69. In our experiment we observed a high cost variance for these RMS values: this is a good indicator that some kind of information might not be captured correctly. Due to intense disk and threading activity, the number of points grows indeed considerably if we take into account external inputs (Figure 6b), and even more if we consider thread-induced inputs as well (Figure 6c). Eventually, we collect 110 distinct points, since each call turns out to have a distinct DRMS value.

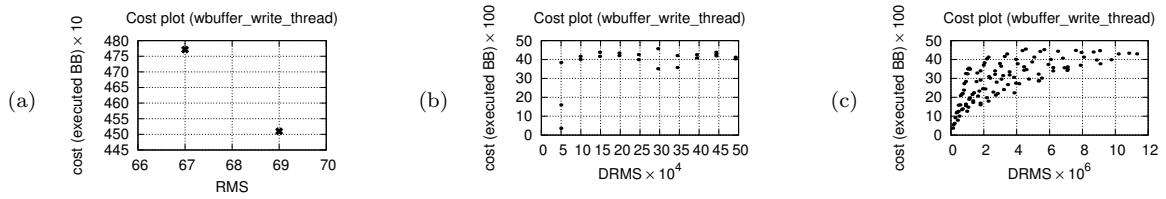

Figure 6: Function `wbuffer_write_thread` of `vips` (PARSEC 2.1): (a) RMS cost plot; (b) DRMS cost plot with external input only; (c) DRMS cost plot with both external and thread input.

3. COMPUTING THE DYNAMIC READ MEMORY SIZE

In this section we describe an efficient algorithm for computing the dynamic read memory size of a routine activation, producing as output a set of performance points that relate the cost of each routine to the observed distinct input sizes. Routine profiles are thread-sensitive, i.e., profiles generated by routine activations made by different threads are kept distinct (if necessary, they can be merged in a subsequent step).

The profiler is given as input multiple traces of program operations associated with timing information. Each trace is generated by a different thread and includes: routine activations (`call`), routine completions (`return`), `read`/`write` memory accesses, and read/write operations performed through kernel system calls (`userToKernel` and `kernelToUser`, necessary to characterize external input).

As a first step, thread-specific traces are logically merged, interleaving operations performed by different threads according to their timestamps, in order to produce a unique execution trace. If two or more operations issued by different threads have the same timestamp, ties are broken arbitrarily: no assumption can be therefore done about which operation will be processed first. We remark that after merge and tie breaking, trace events are totally ordered. For simplicity of exposition, we also assume that `switchThread` events are inserted in the merged trace between any two operations performed by different threads.

For each operation issued by a routine r in a thread t, the profiler must update DRMS and cost information of r with respect to t. Some operations might also require updating profiling data structures related to threads other than t. To clarify the relationships between different threads, we first discuss a naive approach as a warm-up for the reader.

3.1 Naive Approach

According to the definition of dynamic input size (see Section 2), computing $\text{DRMS}_{r,t}$ requires counting read operations issued by routine r that are either first reads or induced first-reads. In turn, identifying induced first-reads requires monitoring write operations performed by *all* threads, i.e., performed also by threads different from t.

A simple-minded approach, which is sketched in Figure 7, is to maintain a set $L_{r,t}$ of memory locations accessed during the activation of r. Immediately after entering r, this set is empty and $\text{DRMS}_{r,t} = 0$. Memory locations can be both added to and removed from $L_{r,t}$ during the execution of r, as follows:

- when r reads or writes a location ℓ, then ℓ is added to $L_{r,t}$ (if not already present);
- when a thread $t' \neq t$ writes a location ℓ, then ℓ is removed from $L_{r,t}$ (if present): this allows recognizing induced first-reads.

Event	Instrumentation (event handler)
$\texttt{read}_t(\ell)$	if $\ell \notin L_{r,t}$ then $\text{DRMS}_{r,t}$++ $L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_t(\ell)$	$L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{read}_{t'}(\ell)$, $t' \neq t$	–
$\texttt{write}_{t'}(\ell)$, $t' \neq t$	$L_{r,t} \leftarrow L_{r,t} \setminus \{\ell\}$

Figure 7: Computation of $\text{DRMS}_{r,t}$ with a naive approach. The notation $\texttt{read}_t/\texttt{write}_t(\ell)$ indicates that location ℓ is read/written by thread t.

With this approach, at any time during the execution of r, a read operation on a location ℓ is a first read (possibly induced by other threads) if and only if $\ell \notin L_{r,t}$. Hence, $\text{DRMS}_{r,t}$ is increased only if this test succeeds. Notice that read operations performed by threads different from t change neither set $L_{r,t}$ nor $\text{DRMS}_{r,t}$.

We remark that in the description above r can be any routine in the call stack of thread t (not necessarily the topmost). Hence, the same checks and updates must be performed for all pending routine activations in the call stack of t. Due to stack-walking and to the fact that write operations require updating sets $L_{r,t}$ of *all* threads, this simple-minded approach is extremely time-consuming. It is also very space demanding: in the worst case, each distinct memory location could be stored in all sets $L_{r,t}$ for each thread t and each routine activation r pending in the call stack of t. In that case, the space would be proportional to the memory size times the maximum stack depth times the number of threads.

3.2 The Read/Write Timestamping Algorithm

To obtain a more space- and time-efficient algorithm, we avoid storing explicitly the dynamic read memory size $\text{DRMS}_{r,t}$ and the sets $L_{r,t}$ of accessed memory locations. Instead, we maintain partial information that can be updated quickly during the computation and from which the DRMS can be easily derived upon the termination of a routine. Capturing the interaction between different threads requires overcoming several technical difficulties, which we sketch in this section. Our algorithm stems from a non-trivial combination of the latest-access approach described in [5] and the use of a novel global timestamping technique.

For each thread t and memory location ℓ, we store ℓ in only one set $P_{r,t}$ such that r is the latest routine activation in t that accessed ℓ (either directly or by its completed subroutines). At any time during the execution of thread t and

```
on_event call(r, t):
 1: count++
 2: top_t ++
 3: S_t[top_t].rtn ← r
 4: S_t[top_t].ts ← count
 5: S_t[top_t].drms ← 0
 6: S_t[top_t].cost ←
        getCost()

on_event return(t):
 1: collect(S_t[top_t].rtn,
        S_t[top_t].drms,
        getCost()-
        S_t[top_t].cost)
 2: S_t[top_t-1].drms +=
        S_t[top_t].drms
 3: top_t--

on_event switchThread():
 1: count++

on_event read(ℓ, t):
 1: if ts_t[ℓ] < wts[ℓ] then
 2:    S_t[top_t].drms ++
 3: else
 4:    if ts_t[ℓ] < S_t[top_t].ts
       then
 5:       S_t[top_t].drms ++
 6:       if ts_t[ℓ] ≠ 0 then
 7:          i = max idx s.t.
                S_t[i].ts ≤ ts_t[ℓ]
 8:          S_t[i].drms --
 9:       end if
10:    end if
11: end if
12: ts_t[ℓ] ← count

on_event write(ℓ, t):
 1: ts_t[ℓ] ← count
 2: wts[ℓ] ← count
```

Figure 8: DRMS **profiling algorithm: thread-induced input.**

for each pending routine activation r, it holds:

$$L_{r,t} = P_{r,t} \cup \{P_{r',t} : \ r' \ descendant \ of \ r\}$$

where r' is any pending routine activation that is above r in the call stack at that time. Sets $P_{r,t}$ will be stored implicitly by associating timestamps to routines and memory locations.

Similarly to the naive approach of Figure 7, locations will be both added to and removed from $P_{r,t}$ to characterize induced first-reads. However, this turns out to be inefficient in a multi-threaded scenario: differently from read operations that change only thread-specific sets, write accesses require changing the sets $P_{r,t}$ of each activation r pending in the call stack of each running thread t. By implicitly updating only one set $P_{r,t}$ per thread, the latest-access algorithm avoids stack walking, but the update time for write accesses is still linear in the number of threads, which can be prohibitive in practice.

To reduce the overhead, we combine the latest access approach with global timestamps that are appropriately updated upon write accesses to memory locations: in this way, we will recognize induced first-reads by comparing thread-specific timestamps with global ones. The entire algorithm is sketched in Figure 8.

Data Structures. The algorithm uses the following global data structures:

- a counter *count* that maintains the total number of thread switches and routine activations for all threads;
- a shadow memory *wts* such that, for each memory location ℓ, $wts[\ell]$ is the timestamp of the latest write operation on ℓ performed by any thread. The timestamp of a memory access is defined as the value of *count* at the time in which the access took place.

Similarly to [5], the algorithm also uses the following thread-specific data structures for each thread t:

- a shadow memory ts_t such that, for each memory location ℓ, $ts_t[\ell]$ is the timestamp of the latest access (read or write) to ℓ made by thread t;
- a shadow run-time stack S_t, whose top is indexed by variable top_t. For each $i \in [1, top_t]$, the i-th stack entry $S_t[i]$ stores:
  - The routine id (rtn), the invocation timestamp (ts), and the cumulative cost ($cost$) of the i-th pending routine activation.
  - The *partial dynamic read memory size* ($drms$) of the activation, defined so that the following invariant property holds throughout the execution for each i such that $1 \leq i \leq top_t$:

$$\forall i, \ 1 \leq i \leq top_t : \quad \text{DRMS}_{i,t} = \sum_{j=i}^{top_t} S_t[j].drms \quad (2)$$

    where $\text{DRMS}_{i,t}$ is a shortcut for $\text{DRMS}_{S_t[i].rtn,t}$. At any time, $\text{DRMS}_{i,t}$ equals the current DRMS value of the i-th pending activation on the portion of the execution trace generated by thread t seen so far.

Invariant 2 implies the following interesting property: for each pending routine activation, its DRMS value can be obtained by summing up its partial dynamic read memory size with the DRMS value of its (unique) pending child, if any. More formally:

$$\text{DRMS}_{top_t,t} = S_t[top_t].drms$$

$$\text{DRMS}_{i,t} = S_t[i].drms + \text{DRMS}_{i+1,t}$$

for each $i \in [1, top_t - 1]$. Hence, if we can correctly maintain the partial dynamic read memory size during the execution, upon completion of a routine we will also get the correct DRMS value.

Algorithm and Analysis. The partial dynamic read memory size can be maintained as shown in Figure 8. We first notice that the global timestamp counter *count* is increased at each thread switch and routine call, and its value is used to update routine timestamps (line 4 of the `call` event handler), global memory timestamps (line 2 of the `write` event handler), and local memory timestamps (lines 1 and 12 of `write` and `read`, respectively). Upon activation of a routine, `call`(r, t) creates and initializes a new shadow stack entry for routine r in S_t. When the routine activation terminates, its cost is collected and its partial DRMS (which at this point coincides with the correct DRMS value according to equation $\text{DRMS}_{top_t,t} = S_t[top_t].drms$ discussed above) is added to the partial DRMS of its parent, preserving Invariant 2.

Local timestamps of memory locations are updated both by read and write accesses, while global timestamps are not updated upon read operations (they are thus associated to write operations only). This update scheme makes it possible to recognize induced first-reads to any location ℓ, which is done by lines 1-2 of `read`. If the read/write timestamp $ts_t[\ell]$ local to thread t is smaller than the global write timestamp $wts[\ell]$, then location ℓ must have been written more recently than the last read/write access to ℓ by thread t. Note that, if the latest access to ℓ was a write operation by thread t, then it would be $ts_t[\ell] = wts[\ell]$ (see the `write` event handler), letting the test $ts_t[\ell] < wts[\ell]$ fail. Hence, if the test succeeds, the last write on ℓ must have been done by some thread $t' \neq t$, the read access by t is an induced access, and the partial DRMS of the topmost routine is correctly

on_event `kernelToUser`(ℓ):
1: $count$++
2: $wts[\ell] \leftarrow count$

on_event `userToKernel`(ℓ, t):
1: `read`(ℓ, t)

Figure 9: DRMS **profiling algorithm: external input.**

increased by line 2 of `read`. Invariant 2 is fully preserved by this assignment: the accessed value is new not only for the topmost routine in the call stack S_t, but also for all its ancestors, whose DRMS is implicitly updated in accordance with Equation 2.

On the other side, if the test of line 1 of `read` fails, the read access to ℓ might still be a first access: this happens if the last access to location ℓ by thread t took place before entering the current (topmost) routine. Lines 4–10 address this case, updating the partial DRMS as described in [5]: the test at line 6 succeeds if and only if location ℓ has been accessed before by thread t during the execution. The latest access happened at some ancestor i of the topmost routine (or in one of its completed descendants): line 7 finds the deepest ancestor i that has accessed ℓ and line 8 decreases its partial DRMS by 1. This restores Invariant 2 for all pending activations below i, whose DRMS must not be affected by the current read operation.

The running time of all operations is constant, except for line 7 of `read` that requires $O(\log d_t)$ worst case time, where d_t is the depth of the call stack S_t.

External Input. So far we have focused on induced first-reads generated by multi-threaded executions. The read-/write timestamping algorithm can be naturally extended to take into account also induced first-reads due to external inputs, relaxing the unrealistic assumption we made in Section 2 that kernel system calls are executed by a separate thread.
Event handlers `userToKernel` and `kernelToUser` shown in Figure 9 update the profiler's data structures when memory accesses are mediated by kernel system calls. Threads invoke system calls to get data from external devices (e.g., disk or network) or to send data to external devices. We remark that the operating system kernel must be treated differently from normal threads in our algorithm, since there are no kernel-specific shadow memory and shadow stack.

When a thread sends data to an external device, it must delegate the operating system to read the memory locations containing those data and write their content to the device. Hence, an OS write operation corresponds to a `userToKernel` event in the execution trace. As shown in Figure 9, read memory accesses by the operating system are regarded as read operations implicitly performed by the thread, as if the system call were a normal subroutine.

The case of `kernelToUser` operations is slightly more subtle. When a thread needs data from an external device, it delegates the operating system to write the device data to some memory buffer. The `kernelToUser` event handler increases *count* and then associates buffer memory locations with a global write timestamp that is larger than any thread-specific timestamp. This forces the test $ts_t[\ell] < wts[\ell]$ to succeed if a buffer location ℓ will be subsequently read by the thread, properly increasing the partial DRMS only for actual read operations.

Counter Overflows. The global counter used by the timestamping algorithm is common to all running threads and in our initial experiments was affected by overflows, especially for long-running applications. Unfortunately, overflows are a serious concern in the computation of the DRMS, since they alter the partial ordering between memory timestamps yielding wrong input size values. To overcome this issue, we perform a periodical global renumbering of timestamps in the profiler's data structures. The main technical difficulty is preserving the partial order between $ts_t[\ell]$, $wts[\ell]$, and $S_t[i].ts$ for each memory location ℓ, running thread t, and $1 \leq i \leq top_t$.

4. EXPERIMENTAL EVALUATION

To prove the feasibiliy of our approach, we implemented the DRMS metric in `aprof` [5], a Valgrind-based open-source profiler. In this section, we discuss the results of an extensive experimental evaluation of the resulting tool, which we call `aprof-drms`, on a variety of benchmarks and we compare it to other prominent heavyweight dynamic analysis tools.

4.1 Experimental Setup

Metrics. Besides slowdown and space overhead, we use the following metrics:

1. *Routine profile richness*: for each routine r, let $|\text{RMS}_r|$ and $|\text{DRMS}_r|$ be the numbers of distinct input sizes collected for routine r by all threads (each value corresponds to a point in the cost plot of r). The profile richness of routine r is defined as:

$$\frac{|\text{DRMS}_r| - |\text{RMS}_r|}{|\text{RMS}_r|}$$

   Intuitively, this metric compares the number of distinct input values obtained using the DRMS and the RMS. We notice that $|\text{DRMS}_r| \geq |\text{RMS}_r|$ does not necessarily hold: it may happen that two distinct RMS values x and y (obtained from two different activations of a routine) correspond to the same DRMS value z, with $z \geq \max\{x, y\}$. Hence, the profile richness may be either positive, if more points are collected using the DRMS, or negative, if more points are collected using the RMS. We will see that in practice the latter case happens quite rarely.

2. *Dynamic input volume*: according to Inequality 1, the DRMS of a routine activation is always larger than or equal to the RMS of the same activation. The dynamic input volume metric characterizes the increase of the input size values due to multi-threading and to external input for an entire execution:

$$1 - \frac{\sum_{\text{routine activations } \langle r,t \rangle} \text{RMS}_{r,t}}{\sum_{\text{routine activations } \langle r,t \rangle} \text{DRMS}_{r,t}}$$

   Values of this metric range in $[0, 1)$. If $\text{DRMS}_{r,t} = \text{RMS}_{r,t}$ for all routine activations r, then the dynamic input volume is 0. Conversely, if $\text{DRMS}_{r,t} \gg \text{RMS}_{r,t}$ for all routine activations r, then the dynamic input volume gets close to 1.

3. *Thread input*: this metric measures the percentage of induced first-reads (line 2 of procedure `read` in Figure 8) due to multi-threading.

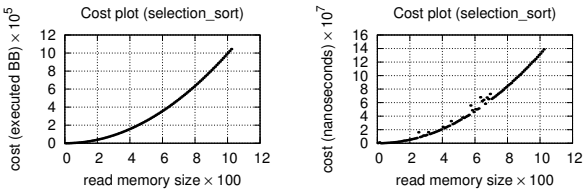

Figure 10: Selection sort: counting basic blocks versus measuring running time.

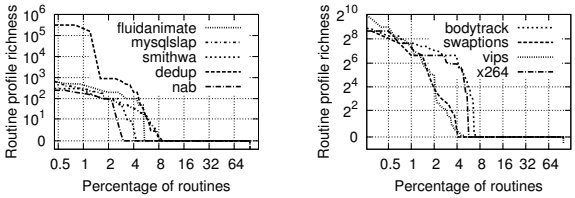

Figure 11: Routine profile richness of DRMS w.r.t. RMS.

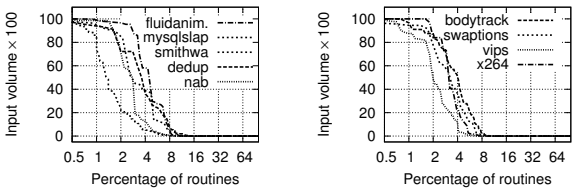

Figure 12: Dynamic input volume of DRMS w.r.t. RMS.

4. *External input*: similarly to the previous case, this metric measures the percentage of induced first-reads due to external input.

Benchmarks. The OMP2012 benchmark suite of the Standard Performance Evaluation Corporation [16] is a collection of fourteen OpenMP-based applications from different science domains. All of them were run on the SPEC `train` workloads.

The Princeton Application Repository for Shared-Memory Computers (PARSEC 2.1) is a benchmark suite for studies of Chip-Multiprocessors [3]. It includes different workloads chosen from a variety of areas such as computer vision, media processing, computational finance, enterprise servers, and animation physics. Experimental results reported in this section are all based on the `simlarge` input sets [3].
We also included in our tests application `MySQL` (version 5.5.30) discussed in Section 2.1: we used the `mysqlslap` load emulation client, simulating 50 concurrent clients that submit approximately 1000 auto-generated queries.

Evaluated Tools. We compared the performance of `aprof-drms` to four reference Valgrind tools: `nulgrind`, which does not collect any useful information and is used only for testing purposes, `memcheck` [17], a tool for detecting memory-related errors, `callgrind` [22], a call-graph generating profiler, and `helgrind` [15], a data race detector. Although the considered tools solve different analysis problems, all of them share the same instrumentation infrastructure provided by Valgrind, which accounts for a significant fraction of the execution times. We also compared `aprof-drms` against the standard version of `aprof`, which is based on the RMS metric.

Implementation Details. `aprof-drms` traces all memory accesses and function calls and returns. Similarly to previous works [5, 8], we use basic blocks as a performance measure. This typically yields the same trends compared to running time measuraments, but is faster and produces neater charts with much lower variance, improving accuracy in characterizing the asymptotic behavior even on small workloads (an example is shown in Figure 10).

To reduce space overhead in practice, we maintain global and thread-specific shadow memories by means of three-level lookup tables, so that only chunks related to memory cells actually accessed by a thread need to be shadowed in its thread-specific memory. To take into account external input, system calls are wrapped and properly mapped to one or more `userToKernel` or `kernelToUser` events: among the main system calls on a Linux x86_64 machine, `write`, `sendto`, `pwrite64`, `writev`, `msgsnd`, and `pwritev` correspond to `userToKernel` events, while `read`, `recvfrom`, `pread64`, `readv`, `msgrcv`, and `preadv` correspond to `kernelToUser` events.

Platform. Experiments were performed on a cluster machine with four nodes, each equipped with two 64-bit AMD Opteron Processors 6272 @ 2.10 GHz (32 cores), with 64 GB of RAM running Linux kernel 2.6.32 with `gcc` 4.4.7 and Valgrind 3.8.1 – SVN rev. 13126.

4.2 Experimental Results

The goals of our experiments are threefold: evaluating the benefits of the DRMS, showing that `aprof-drms` can characterize the nature of dynamic workloads, and assessing its slowdown and space overhead.

DRMS versus RMS. As shown in [5], an RMS-based profiler can collect a significant number of distinct input sizes for most algorithmic-intensive functions. A first natural question is whether using DRMS instead of RMS has any impact on the profile richness. Charts in Figure 11 contribute to answer this question, focusing on a representative set of benchmarks. A point (x, y) on a curve means that x% of routines have profile richness at least y: e.g., in benchmark `dedup`, the number of points collected by the DRMS is more than 100 times larger than using the RMS for roughly 4% of the routines. As expected, only a small percentage of routines has high values of profile richness, since I/O and thread communication are typically encapsulated in a few software components. However, for these routines $|\text{DRMS}_r|$ can be substantially larger than $|\text{RMS}_r|$ (up to a factor of roughly 10^6 for benchmark `dedup`). We also notice that only a statistically intangible number of routines has negative profile richness: this means that the DRMS can almost always generate plots with more points than the RMS.

Due to Inequality 1, DRMS values are always larger that RMS values for the same routine activations. Figure 12 characterizes the increase of the input size values due to induced first-reads. The interpretation of these graphs is similar to Figure 11: a point (x, y) on each benchmark-specific curve means that x% of routines have dynamic input volume $\geq y$. For instance, in benchmark `fluidanimate`, roughly 3% of

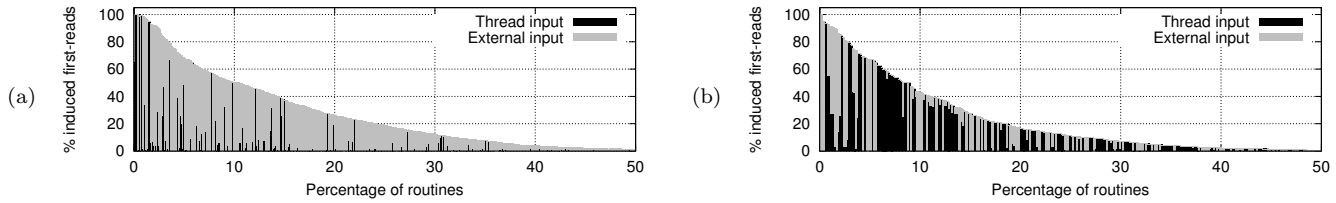

Figure 13: Routine-by-routine thread and external input on benchmarks (a) `MySQL` and (b) `vips`.

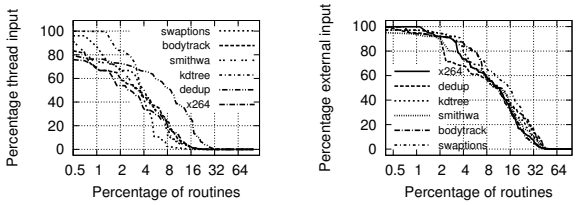

Figure 14: Thread and external input on a routine basis.

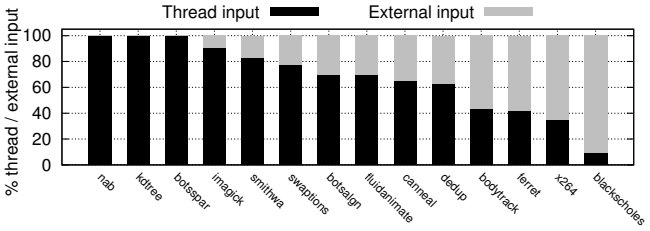

Figure 15: Characterization of induced first-reads.

the routines take almost all their input from external devices or from other threads. The trend of curves in Figure 12 decreases steeply from 100 to 0, reaching its minimum at $x \simeq 8\%$ for most benchmarks: this means that 8% of the routines are responsible of thread intercommunication and streamed I/O, and the input size of these routines cannot be appropriately predicted by the RMS metric alone.

Dynamic Workload Characterization. Besides producing estimates of the empirical cost functions of routines, input measurement metrics also yield a characterization of the typical workloads on which routines are called in the context of deployed systems [5]. In particular, rich profile data collected by the DRMS metric can provide insights on the amount of interaction with external devices (external input) and cooperating threads (thread input). Charts in Figures 13, 14, and 15 characterize the workload of an application at different levels of granularity and can be automatically produced by our profiler.

If we sort in decreasing order all routines by percentage of induced first-reads, we can assess the interplay between workload, computation, and concurrency, as shown in Figure 13. For each routine of benchmarks `MySQL` and `vips`, the histogram plots the percentage of induced first-reads partitioned between thread and external input. A first look reveals that induced first-reads of the majority of `MySQL` routines are due to external input, confirming the fact that `MySQL` makes extensive use of both network and I/O. Conversely, thread input turns out to be predominant in `vips`, which is indeed a data-parallel image processing application. Figure 14 compactly represents the percentage of thread and external input over the total number of (possibly induced) first-read operations: a point (x, y) on each benchmark-specific curve means that x% of routines have external / thread input $\geq y$%. For instance, in benchmark `dedup`, 16% of the routines are such that at least 20% of their first-reads are due to thread intercommunication. By computing the percentages w.r.t. the total number of induced first-reads, we can instead obtain the histogram in Figure 15, where each bar sums up to 100%. Benchmarks are sorted by decreasing thread input (and thus by increasing external input). An interesting observation is that the SPEC OMP2012 benchmarks get naturally clustered in the leftmost part of the histogram (from `nab` to `botsalgn`), and all of them have thread input larger than 69%.

Notice that the execution of an application may change according to the thread interleaving chosen by a scheduler. An interesting aspect is how this choice can affect the shared-memory communication between threads and thus the DRMS metric. We analyzed several runs of different applications taken from the SPEC OMP2012 and PARSEC 2.1 benchmark suites using multiple Valgrind's scheduling configurations. As expected, the external input remains stable across different runs. On the other hand, thread input shows a mean fluctuation of less than 2%, with peaks up to 800% for a few benchmarks. This alteration, however, does not qualitatively affect the observed trends in the routine cost plots.

Slowdown and Space Overhead. Performance figures on the SPEC OMP2012 and PARSEC 2.1 benchmark suites, obtained spawning four threads per benchmark, are summarized in Table 1 (see [6] for detailed results). Compared to native execution, all the evaluated tools exhibit a large slowdown: even `nulgrind`, which is reported to be roughly 5 times slower than native [20], in our experiments turned out to have mean slowdown factors of 23.6× and 12.2× respectively for the two suites. `aprof-drms` is on average 6 times slower than `nulgrind`. The slowdown is worse than `memcheck`, which is 1.5 times faster than our tool but does not trace function calls and returns. `helgrind`, the only tool designed for the analysis of concurrent computations, results to be slower than `aprof-drms` (e.g., 2.2 times on the PARSEC 2.1 suite). Recognizing induced first-reads causes an average overhead of 29% on the running time, as demonstrated by the comparison of `aprof-drms` with `aprof`.
The mean memory requirements of `aprof-drms` are within a factor of 3.3× on SPEC OMP2012 and of 6.1× on PARSEC 2.1: this overhead mostly depends on shadow memories. `memcheck` turns out to be more efficient than our tool

	nulgrind	memcheck	callgrind	helgrind	aprof	aprof-drms
Slowdown (Geom. Mean)						
SPEC OMP	23.6×	94.1×	64.8×	179.4×	101.5×	140.8×
PARSEC 2.1	12.2×	51.8×	51.4×	153.3×	57.1×	68.2×
Space overhead (Geom. Mean)						
SPEC OMP	1.4×	2.0×	1.5×	4.5×	2.8×	3.3×
PARSEC 2.1	1.8×	2.9×	2.1×	8.4×	4.6×	6.1×

Table 1: Performance comparison of `aprof-drms` with `aprof` and some prominent Valgrind tools.

thanks to the adoption of memory compression schemes and to its independence from the number of threads. Similarly, `aprof` is slightly more efficient than our tool due to the lack of a global shadow memory. On the other hand, `helgrind`, which is akin to our tool with respect to the analysis of concurrency issues, uses 36% more space than `aprof-drms`. We remark that benchmarks of the PARSEC 2.1 suite use very small memory on the `largesim` workload and thus the overhead is very high even for `nullgrind` and `callgrind` which do not use shadow memories.

Figure 16 shows the average slowdown and space overhead, with respect to the native execution, as a function of the number of spawned OpenMP threads for the SPEC OMP2012 suite. Due to Valgrind thread serialization, on parallel benchmarks such as OMP 2012 the slowdown of all Valgrind tools increases with the number of threads. This is an issue of the Valgrind instrumentation infrastructure, which does not exploit multiple cores, rather than of Valgrind tools, including `aprof-drms`. Regarding space overhead, we observe a modest growth when the number of threads increases, but the memory requirement of `aprof-drms` remains always smaller than `helgrind`, confirming that the performance of our tool is comparable to other heavyweight tools.

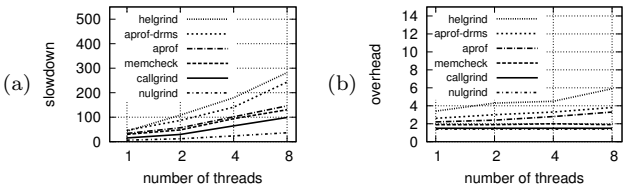

Figure 16: (a) Time and (b) space overhead as a function of the number of threads.

5. RELATED WORK

There is a vast literature on performance profiling, both at the inter- and intra-procedural level: see, e.g., [1, 2, 4, 9–11, 21, 24] and the references therein. All these works aim at associating performance metrics to distinct paths traversed in the call graph or in the control flow graph during a program's execution. Input-sensitivity issues are instead explored in [5, 8, 13, 23]. Marin and Mellor-Crummey [13] consider the problem of understanding how an application's performance scales given different problem sizes, using data collected from multiple runs with determinable input parameters. Goldsmith, Aiken, and Wilkerson [8] also propose to run a program on workloads of different sizes, to measure the performance of its routines, and eventually to fit these observations to a model that predicts how the performance scales. The workload size of the program's routines, however, is not computed automatically. Algorithmic profiling by Zaparanuks and Hauswirth [23], besides identifying boundaries between different algorithms in a program, infers their computational cost, which is related to the input size. The notion of input size is defined at a high level of abstraction, using different definitions for different data structures (e.g., the size of an array or the number of nodes in a tree). The input-sensitive profiling methodology described in [5], which provides the basis for our approach, automatically infers the input size by tracing low-level memory accesses performed by different routines. None of these approaches addresses dynamic workloads, ignoring exernal inputs and communication among threads.

The problem of empirically studying the asymptotic behavior of a program has been the target of extensive research in experimental algorithmics [7, 14], where individual modules are extracted from applications and separately analyzed on ad-hoc test harnesses. However, by studying performance-critical routines out of the context of the overall application in which they are deployed, this approach may fail to characterize their actual behavior in production environments.

6. CONCLUSIONS

In this paper we have extended the input-sensitive profiling methodology [5] in order to include dynamic input sources such as communication between threads and I/O. In more details, we have proposed a novel metric, called drms, that gives an estimate of the size of dynamic workloads of each routine activation by taking into account first read operations, possibly induced by other threads or by kernel system calls.

As a future direction, it would be interesting to port our implementation to a fully scalable and concurrent dynamic instrumentation framework, in order to exploit parallelism to leverage the overhead of our profiler. Our approach also raises interesting open issues regarding input characterization and thread intercommunication in concurrent applications. In a recent experimental study [12], it has been observed that even widespread multi-threaded benchmarks do not interact much or interact only in limited ways, and that communication does not change predictably as a function of the number of cores. We believe that our drms computation algorithm may support the development of automatic tools for characterizing how multi-threaded applications scale their work and how they communicate via shared memory at routine activation rather than thread granularity.

7. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

[2] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, pages 168–179. ACM, 2001.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *17th Int. Conference on Parallel Architecture and Compilation Techniques*, pages 72–81, 2008.

[4] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *CGO*, pages 205–216. IEEE Computer Society, 2005.

[5] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI*, pages 89–98, 2012.

[6] E. Coppa, C. Demetrescu, I. Finocchi, and R. Marotta. Multithreaded input-sensitive profiling. Technical report, 2013.

[7] C. Demetrescu, I. Finocchi, and G. F. Italiano. Algorithm engineering. *Bulletin of the EATCS (algorithmics column)*, 79:48–63, 2003.

[8] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE*, pages 395–404, 2007.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler (with retrospective). In K. S. McKinley, editor, *Best of PLDI*, pages 49–57. ACM, 1982.

[10] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proc. Summer 1993 USENIX Technical Conference*, pages 1–19, 1993.

[11] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[12] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *OOPSLA*, pages 335–354, 2012.

[13] G. Marin and J. M. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS*, pages 2–13, 2004.

[14] C. C. McGeoch, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. Using finite experiments to study asymptotic performance. In *Experimental Algorithmics*, LNCS 2547, pages 93–126, 2002.

[15] A. Mühlenfeld and F. Wotawa. Fault detection in multi-threaded C++ server applications. *Electron. Notes Theor. Comput. Sci.*, 174(9):5–22, 2007.

[16] M. S. Müller and *et al.* SPEC OMP2012 – an application benchmark suite for parallel systems using OpenMP. In *Proc. 8th Int. Conf. on OpenMP in a Heterogeneous World*, pages 223–236, 2012.

[17] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[18] J. M. Spivey. Fast, accurate call graph profiling. *Softw., Pract. Exper.*, 34(3):249–264, 2004.

[19] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 2006.

[20] Valgrind tool suite. `http://www.valgrind.org/info/tools.html`.

[21] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL*, pages 351–362. ACM, 2007.

[22] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447, 2004.

[23] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, pages 67–76, 2012.

[24] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.