Int J Parallel Prog (2012) 40:410–442
DOI 10.1007/s10766-012-0193-x

Trin-Trin: Who's Calling? A Pin-Based Dynamic Call Graph Extraction Framework

Rohit Jalan · Arun Kejariwal

Received: 15 November 2011 / Accepted: 6 April 2012 / Published online: 12 May 2012

Abstract Multi-core based systems are ubiquitous in data centers. Efficient exploitation of hardware parallelism supported by such systems is imperative on multiple fronts: minimizing latency and power consumption and maximizing throughput. This in turn calls for advanced program analysis and optimization. Call graphs have been long used to this end. Although several static call graph extraction techniques have been proposed in the past, these techniques cannot be applied to analyze programs already running in production. Likewise, the existing dynamic call graph extraction tools have limited use in production owing to, say (but not limited to), lack of support for capturing wall clock time spent in functions of a given program and lack of means to analyze the call graph information captured at run time. In this paper, we present a Pin-based dynamic call graph extraction framework called **Trin-Trin**. The framework enables extraction of complete, precise and dynamic call graphs. Additionally, the framework can be used seamlessly with already running applications. Furthermore, an analytics engine is provided to facilitate advanced program analysis, e.g., different multithreading context(s) of any function can be extracted in a demand-driven fashion. We evaluate the overhead of **Trin-Trin** using several Unix utilities, applications from the industry-standard SPEC CINT2006, CFP2006 benchmark suite and Yahoo! properties. Additionally, we present a case study to illustrate how **Trin-Trin** can be used to analyze performance bottlenecks and performance regressions.

Keywords Performance · Profiling · Call Graphs · Analytics · Multithreading · Multi-cores

R. Jalan · A. Kejariwal (✉)
Yahoo! Inc., Sunnyvale, CA, USA
e-mail: arun_kejariwal@acm.org

1 Introduction

Low latency is one of the key metrics associated with user experience in the Internet space. From a corporate standpoint, it is equally critical to serve a large audience (i.e., support high throughput) as it directly relates to the bottomline. Achieving the aforementioned dual objectives involves a multi-level process. One of the steps in this regard is program optimization. For example, let us consider the online property Yahoo! Finance (websites such as Yahoo! Finance, Yahoo! Sports are referred to as the different properties of Yahoo!). Optimizing the property for lower latency directly relates to better user experience.

Program optimization has been a subject of research for over four decades [1,32]. Several static and dynamic program optimization approaches have been proposed. In most approaches, construction of precise whole program call graphs [2,7,24,29,49] is required. Call graphs have been used to guide interprocedural optimization [5], guide the replacement of dynamically dispatched function calls with direct method calls, guide function inlining et cetera. Further, call graphs have been used in reverse engineering of software systems [10,11]. Additionally, it has been shown that call graph construction has ramifications on the precision of termination checking [50].

Several static call graph extraction approaches have been proposed. The preciseness of these approaches is limited owing to, say, the use of function pointers, indirect referencing. This in turn limits guiding, say, function inlining and interprocedural optimization. Further, by definition, these approaches do not shed any light on the coverage—defined as the percentage of the run time—of a given function. To this end, commercial and open source dynamic call graph extractors such as Intel's `vtss-run` [27], `Callgrind` [8] and `Gprof` [19] have been developed. Function coverage reported by the existing tools is based on CPU cycles instead of the wall clock time/run time. Consequently, the percentage of run time spent in kernel space by applications routines cannot be quantified via these tools (e.g., on 32-bit platforms, the amount run time spent in the kernel space can be approximated by the coverage of the `_dl_sys-info_int80` routine[1]). This can be severely limiting in context of multithreaded and I/O intensive applications wherein a large part of a program's run time may be spent in the kernel space. We overview existing static and dynamic call graph extraction approaches in Sect. 6.

To address the above limitations, we present **Trin-Trin**—a framework to extract *complete, precise and dynamic* call graphs. In the current context, *completeness* signifies that **Trin-Trin** also captures the run time spent in the kernel space. For this, **Trin-Trin** measures processor cycles (using the `rdtsc` instruction) for each thread. The time spent by a thread in the kernel space can be approximated by summing the run times of routines that are known to context switch into the kernel. This can potentially be of help to characterize the multithreaded execution behavior of industry-standard benchmarks such as SPEC OMP2001 [58] and to assess the efficacy of different thread synchronization strategies. *Preciseness* signifies that, in contrast to static call graph extractors, *no* approximations are made with respect to caller–callee relationships.

[1] The routine `_dl_sysinfo_int80` taps into the kernel to invoke system services.

Lastly, *dynamic* signifies that the extracted call graph corresponds to the program flow at run time; in other words, functions that are not called at run time do not feature in the extracted call graph. **Trin-Trin** can also capture the calls to functions in the dynamic linker. **Trin-Trin** is based on Intel's Pin dynamic instrumentation system [31]. We have been using **Trin-Trin** to analyze the run time performance of multiple Yahoo! properties such as the Y! search engine and the advertising platform.

The main contributions of the paper are as follows:

- First, we present a framework—**Trin-Trin**—to extract *complete, precise and dynamic* call graphs. **Trin-Trin** can be used not only for sequential, but also for multithreaded as well as multi-process applications. In the case of a multi-process application, **Trin-Trin** generates a separate call graph for each process. A key highlight of **Trin-Trin** is that it can be used to extract call graphs of already running applications. This is of particular importance as applications in production cannot be restarted frequently. A pid-based attachment/detachment mode is supported in **Trin-Trin** for the above. The design and features of **Trin-Trin** are detailed in Sect. 3.
- Second, we present an analytics engine to assist a developer and/or a performance engineer with advanced program and performance analysis. The analytics engine can be used to determine, for example, hottest path in a call graph, existence of cycles in a call graph, depth of recursion, levels of multithreading, the number of multithreaded contexts a function was called in et cetera. The analytics engine supports graphical visualization, using the open source `dot` format [20], of the extracted call graphs. All the call graph illustrations presented in the rest of the paper were generated using the analytics engine. Note that the analytics engine is run post-collection of the call graph profile data. Thus, the analytic engine does *not* introduce any run time overhead. The analytic engine is discussed in detail in Sect. 4.
- Third, we evaluate the overhead of **Trin-Trin** using several Unix utilities such as `find`, applications from the industry-standard SPEC CINT2006 [57], CFP2006 [56] benchmark suite.
- Fourth, we present a case study to illustrate how **Trin-Trin** can be used to analyze performance bottlenecks and performance regressions.

The rest of the paper is organized as follows: Sect. 2 introduces the terminology used in the rest of the paper. Section 3 details the design on **Trin-Trin**. Section 4 presents the analytics engine. Experimental results—run time overhead of **Trin-Trin** and illustration of its use in program optimization—are presented in Sect. 5. Previous work is discussed in Sect. 6. Finally, in Sect. 7, we conclude with directions for future work.

2 Terminology

In this section, we introduce the terminology used in the rest of the paper. A call graph *G(V, E)* is a directed graph where V is a finite set of nodes and E is a binary relation on V. Given two nodes u, v with an edge between them $u \rightarrow v$, we say that u is the source and v is the sink.

```
void foo(int level, int delay) {
    if (level == 1)      { sleep(delay); bar(1, delay); }
    else if (level == 2) { sleep(delay); toy(1, delay); }
    else                 { sleep(delay); toy(2, delay); }
}
void bar(int level, int delay) {
    if (level == 1)      { sleep(delay); foo(2, delay); }
    else if (level == 2) { sleep(delay); foo(3, delay); }
    else                 { sleep(delay); toy(3, delay); }
}
void toy(int level, int delay) {
    if (level == 1)      { sleep(delay); bar(2, delay); }
    else if (level == 2) { sleep(delay); bar(3, delay); }
    else { sleep(delay); }
}
int main(int argc, char **argv)
{
    foo (1, 1); return 0;
}
```

Fig. 1 Example program

Definition 1 A **path** from a node u to a node v in a call graph G(V, E) is a sequence of nodes $\langle n_0, n_1, \ldots, n_k\rangle$ such that $u = n_0, v = n_k$ and $(n_{i-1}, n_i) \in E$ for $i = 1, 2, \ldots, k$.

Definition 2 In a call graph G(V, E), a path $\langle n_0, n_1, \ldots, n_k\rangle$ forms a **cycle** if $n_0 = n_k$ and the path contains at least one edge. The cycle is **simple** if $n_0, n_1, \ldots, n_k$ are distinct, else the cycle is referred to as a **complex** cycle. A **self-loop** is a simple cycle with only one node.

Note that existence of simple/complex cycle(s) in a call graph signifies the use of recursion in the corresponding program. Let us consider the example program shown in Fig. 1.

Figure 2a shows a partial call graph—with the function `sleep` and its callees—of the example program shown in Fig. 1. From the call graph we note that there are two intermediate nodes starting with '__' between `sleep` and the leaf nodes. Typically, such nodes have low coverage (defined as the percentage of overall run time) and hence, are filtered from the graph. However, removal of these nodes orphans the leaf nodes, as shown in Fig. 2b. To this end, we introduce *dfa* edges to connect the leaf nodes with `sleep`, as shown in Fig. 2c. Formally, a *dfa* edge is defines as follows:

Definition 3 A dfa(n) edge $u \rightarrow v$ represents a path between the nodes u and v, n denotes the number of edges between the source and the sink of the dfa edge. dfa stands for 'distance from ancestor'.

The different scenarios in which *dfa* edges are drawn are discussed in Sect. 4.

3 Trin-Trin: Design and Implementation

Broadly, the design of **Trin-Trin** has been guided by the following objectives:

- Assist performance engineers in identifying performance bottlenecks which have the most bang-for-the-buck. In this regard, **Trin-Trin** addresses the following:

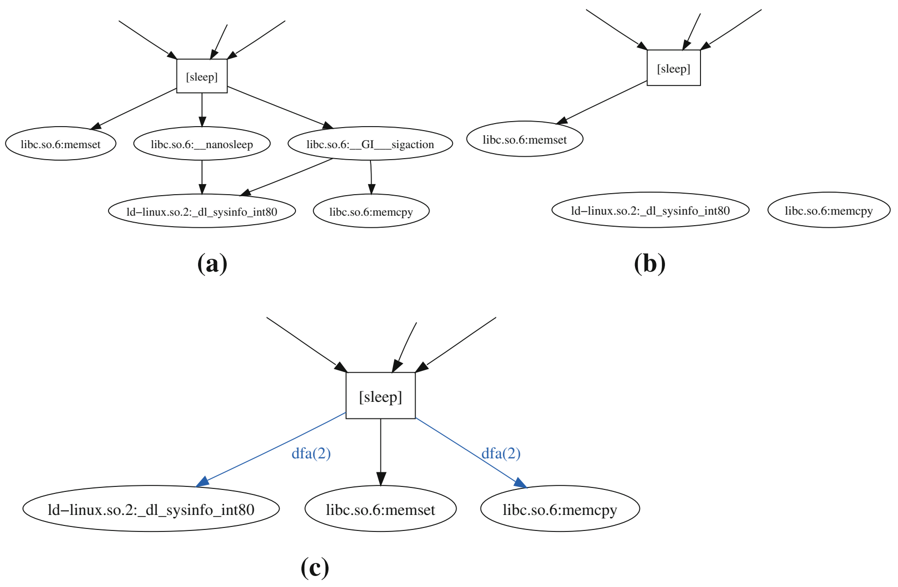


Fig. 2 Illustrating the use of the *dfa* edges

- – What is the coverage, defined as the percentage of the run time, of each function—on a per-thread basis—in a given application?
- – What percentage of the run time is spent by an application sleeping, waiting or executing particular system calls?
- – What is the relative contribution of specific functions to a latency sensitive path in an application?
- – What are the benefits/drawbacks associated with a particular I/O or computation strategy?

- Guide program optimization and selection of compiler optimizations. For example, **Trin-Trin** can be used to guide, by leveraging the context information, how many helper threads [63] should be spawned.

Trin-Trin has been engineered to be portable across 32 and 64-bit x86 platforms and it leverages Pin toolkit capabilities to allow seamless profiling of 32-bit binaries running in a 64-bit environment.

Figure 3 shows the integration of **Trin-Trin** with Pin. As shown in the figure, there are 5 modules in **Trin-Trin**. Each module interacts with Pin by calling, as needed, the Pin APIs. In the rest of this section, we detail each module of **Trin-Trin**.

3.1 Shared Library and Routine Filter

Modern software architecture paradigms advocate splitting code into smaller reusable functions. Thus, real-life programs such as Y! properties consist of a large number of functions. The above, in the context of extraction of dynamic call graphs, leads to the

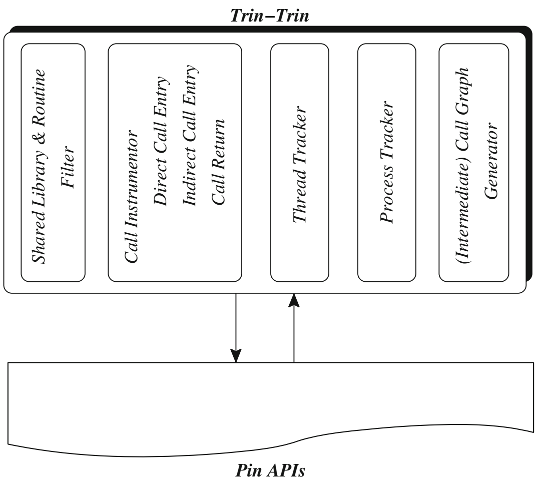

Trin–Trin
Shared Library & Routine Filter
Call Instrumentor Direct Call Entry Indirect Call Entry Call Return
Thread Tracker
Process Tracker
(Intermediate) Call Graph Generator
Pin APIs

Fig. 3 **Trin-Trin**'s integration with Pin

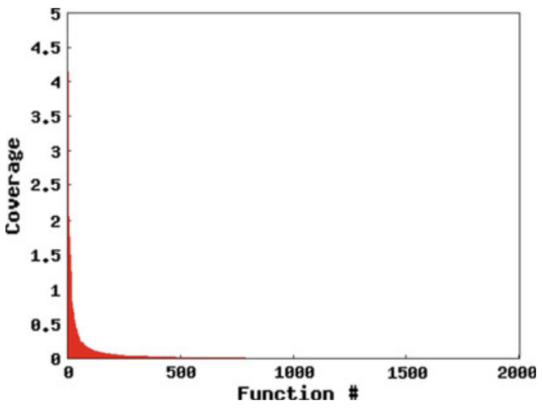


Fig. 4 `403.gcc` function coverage profile

following twofold problems: (a) high run time overheads and (b) dynamic call graphs that are difficult to analyze owing to a large number of nodes.

The aforementioned problems can be mitigated during call graph analysis by filtering nodes and/or edges in the call graph based on their coverages. The Analytics Engine (discussed in Sect. 4) supports selection of, via use of function and/or library name regular expressions, a subset of profiled nodes to be included in the final dynamic call graph. However, this does not address the high extraction overhead problem which can be significant for certain classes of applications.

For example, the function coverage profile of `403.gcc`—a benchmark in the industry-standard SPEC CINT2006 suite [57]—is shown in Fig. 4. The coverage profile was obtained via **Trin-Trin** while running `403.gcc` on a real machine (see Table 2) with the reference data set. From the figure we note that the run time is primarily distributed among 750 functions (from a total of 2,000 functions executed at run time). Plotting a call graph for `403.gcc` will lead to >1,000 nodes in the call graph having very low coverages and the resultant call graph would not be amenable to program analysis.

Likewise, the `483.xalancbmk` benchmark [57] which executes over 10B (!) dynamic calls when run with the reference data set. High dynamic call counts result in high run time overheads while extracting dynamic call graphs.

In some scenarios such as mapping interaction between two logical components of an application or profiling I/O behavior of an application et cetera, it may be beneficial to generate call graphs for specific parts of a given application.

To this end, **Trin-Trin** enables profiling of calls originating from a set of pre-selected functions. This feature has been provided in the form of a regular expression filter that is applied to names of routines containing the call when inserting analysis call backs. By default, **Trin-Trin** uses the '*' filter which profiles all calls in an application.

The filtering feature described above is very useful for limiting call graph extraction overhead when profiling the behavior of high performance and latency sensitive applications such as Y! Finance. In particular, to gauge the latency for a search request on Y! Finance, filtering can be used so as to profile calls corresponding to the top level search request method in a HTTP server application, thereby creating a relatively lesser impact on run time performance.

3.2 Call Instrumentor

Extraction and analysis of dynamic call graphs necessitates tracing the call flow and capturing the dynamic call metadata. The former involves detection and capture of the call sequences. The latter entails capturing metadata such as (but not limited to) call count and coverage on a per call basis. **Trin-Trin**'s modules *Thread Tracker* and *Process Tracker* (discussed subsequently in this section) facilitate call flow tracing and metadata capture of multithreaded and multi-process applications respectively. Call flow tracing support is provided by instrumenting (selected) call instructions in an application. As shown in Fig. 5, **Trin-Trin** inserts Pin's analysis call backs before and after the selected call instructions.

Fig. 5 Example call instrumentation

```
00001000 <A>:
    1000:   b8 0a 00 00 00          mov    $0xa,%eax
    /* Trin-Trin PRE-CALL call back */
    1005:   ff 1d 0e 20 00 00       lcall  *0x200e
    /* Trin-Trin POST-CALL call back */
    100b:   5d                      pop    %ebp
    100c:   01 d8                   add    %ebx,%eax
    ...

0000200e <B>:
    200e:   83 c3 32                add    $0x32,%ebx
    2011:   ff 25 17 30 00 00       jmp    *0x3017
    ...

00003017 <C>:
    3017:   01 c3                   add    %eax,%ebx
    /* Trin-Trin PRE-CALL call back */
    3019:   ff 1d 20 40 00 00       lcall  *0x4020
    /* Trin-Trin POST-CALL call back */
    301f:   c3                      ret
    ...

00004020 <D>:
    4020:   89 c8                   mov    %ecx,%eax
    4022:   c3                      ret
```

Instrumentation of call instructions is not trivial owing to the following two reasons. First, during instruction analysis, the Pin API function `INS_InsertCall` can only insert instrumentation before a call instruction. Second, Pin's instruction analysis order is not linear with respect to the instructions contained in an application binary; instead, it is a function of the program flow. We address the first constraint by instrumenting each call instruction with a `IPOINT_BEFORE` call back (in other words, with a pre-call call back). As mandated by the Pin API, `IPOINT_BEFORE`, pre-call call back is called before the call instruction is executed. Instructions occurring after the selected call instruction are also instrumented with a corresponding `IPOINT_BEFORE` call back. This call back serves as a complimentary post-call call back for the previously inserted pre-call call back. The second constraint is addressed by tracking the insertion of pre- and post-callbacks for each selected call instruction. This methodology allows us to insert call instrumentation asynchronously. Specifically, it allows us to address the situation where multiple pre-call call backs are inserted before their corresponding post-call call backs are inserted.

Call metadata capture is carried out within the context of pre- and post-call call backs. Currently, **Trin-Trin** tracks recursive and non-recursive edge call counts, function entry and exit timestamps and minimum and maximum call durations for each traced function (refer to Fig. 7).

3.2.1 Handling Call Source Addresses

By design, **Trin-Trin** uses the destination address of the last call as the source address of the next call during flow tracing. This is done to avoid creating unconnected nodes in the call graph. Consider the example given in Fig. 5. From the figure we note that function `A` located at address *0x1000* calls function `B` from position *0x1005*. `B` located at address *0x200e* then jumps to procedure `C` from position *0x2011*. `C` located at *0x3017* calls the function `D` from position *0x3019* and then returns to its caller. If we trace the call edges using actual source addresses the call sequence is `A` → `B`, `C` → `D` and the resultant call graph is shown in Fig. 6a.

Unlike **Trin-Trin**, a näive strategy of using unadjusted call source and destination addresses results in a large number of top-level (unconnected) nodes in a dynamic call graph. This has direct implications on call graph analysis. **Trin-Trin** produces the following call sequence: `A` → `B` → `D`; the resulting call graph is shown in Fig. 6b.

In general, tracing `'JMP'` instructions in applications is not a viable option due to its to dramatic impact on run time overhead. Therefore, we decided to adopt the

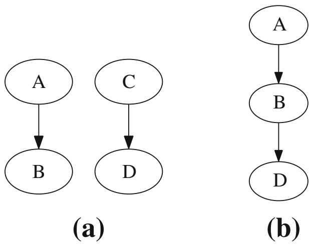


Fig. 6 Sample call graphs for example presented in Fig. 5

```
typedef struct edge_stat {          typedef struct rcr_nodeinfo {
  edge_t es_edge;                     /* current recursion level of node */
  uint64_t es_total_duration;         int rcr_cur_level;
  uint64_t es_max_duration;           /* occurrence count of node */
  uint64_t es_min_duration;           int rcr_cur_ecount;
  uint64_t es_count;                  /* pointer to last occurrence */
  uint64_t es_rcr_count;              int rcr_last_idx;
  rcr_nodeinfo_t *es_rcr;             ...
  ...
```

Fig. 7 Metrics tracked by **Trin-Trin**

approach mentioned above. Note that this approach does not require mapping of call source address to the entry point of functions in which they are contained. For example, to produce a correct call graph, we need not explicitly translate the source address of the call at *0x1005* to the address of function `A` (*0x1000*).

3.2.2 Handling Recursion

Trin-Trin's analysis call backs are designed to detect recursion during program execution. Broadly, we classify recursive execution into two classes: (a) *Self Recursion* corresponds to the case when there exists at least one simple cycle with only one node and (b) *Indirect Recursion* corresponds to the case when there exists at least one simple/complex cycles with two or more nodes.

Trin-Trin records recursion information on a per node basis (refer to Fig. 7). Specifically, for each node, there are two metrics associated with recursion—'Call Stack Back Edge Count' and 'Call Stack Node Occurrence Count'. Back edge count is used to calculate the recursion level of a thread; its default value is 0 and is incremented every time a recursive call is made. Node occurrence count also has a default value of 0 and is incremented each time a particular function is invoked. A node occurrence count value greater than one at any recursion level signifies the start of a new recursive call.

Support for detecting recursion at run time has an adverse effect on call metadata capture overhead. Hence, recursion detection is made configurable via a compile-time option in **Trin-Trin**. Contemporary call graph tools do not provide such a feature. For example, Callgrind [8] avoids this problem by stripping cyclic paths from generated call graphs, thereby resulting in incomplete call graphs for recursive programs.

3.3 Thread Tracker

Trin-Trin supports extraction of dynamic call graph of multi-threaded applications. At run time, for each thread, the call information is stored in memory dedicated to a given thread. This reduces, in some cases eliminates, the need of locking call metadata capture data structures across threads. The above strategy also helps in avoiding false cache-line-sharing between threads. At first, call graphs are produced on a per-thread basis, thereby facilitating analysis of per-thread call graphs in isolation, if required. The per-thread call graphs are subsequently "merged" together by the Analytics Engine (discussed in the next section).

Along with monitoring Pin's thread-specific callbacks, **Trin-Trin** also detects and monitors thread creation functions such as `pthread_create` et cetera. This is done via a simple routine name matching algorithm which is triggered during Pin's static routine analysis phase (viz., Routine Analysis Call Back). Information gathered from these three types of call backs is used to relate threads to each other via the parent-child relationship and to detect thread creation points in a program.

3.4 Process Tracker

The *process tracker* module is responsible for tracking child processes. Multi-process applications start as a single process and subsequently spawn new processes using the 'fork()' Unix system call. **Trin-Trin** registers call backs with Pin to track process creation. When a new process is created, **Trin-Trin** re-initializes (resets) call accounting information for the new process. This is required to avoid double accounting of parent process metrics in child process(es). Akin to the multithreaded case, **Trin-Trin** generates individual call graphs for all threads in child processes.

3.5 Intermediate Call Graph Generation

Trin-Trin facilitates extraction of dynamic call graphs at *arbitrary* points during program execution. The intermediate call graphs filenames are suffixed with a version counter that is incremented each time a call graph is generated. The Analytics Engine (discussed in Sect. 4) is capable of recognizing this versioning scheme and it performs analysis on different versions separately. At present we do not correlate data between different versions and each version is analyzed in isolation.

Providing support for generation of intermediate call graphs results in a run time situation in which dynamic call metadata is accessed simultaneously by the *Intermediate Call Graph Generator* and *Call Instrumentor* modules. To avoid corruption of the dynamic call metadata, we introduced locking in our otherwise lock-less call analysis strategy, thereby increasing the overhead of **Trin-Trin**. We decided to keep this as the default behavior with the compile time option of disabling the aforementioned locking, thereby avoiding the overhead.

The *Intermediate Call Graph Generator* module tracks signals, such as `SIGUSR2`, sent to the application being profiled. Specifically, upon receipt of a particular signal (a configurable parameter), **Trin-Trin** produces an intermediate call graph.

An important use case of the *Intermediate Call Graph Generator* module is the extraction of dynamic call graphs of server applications such as the various Y! properties that run for indefinite durations as high availability is key to user experience. Furthermore, the module is also useful in cases where the target application is known to terminate abnormally after a certain duration. In such a scenario, the errant application can be sent a signal to generate a call graph prior to its abnormal termination.

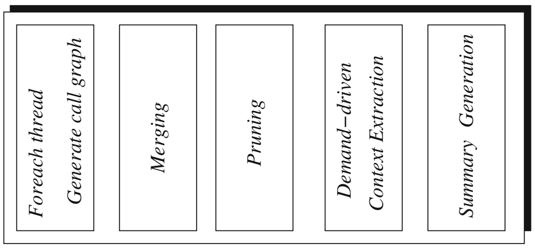


Fig. 8 Analytics engine pipeline

3.5.1 Attach Mode

Based on Pin's support for attaching to running processes, **Trin-Trin** supports profiling of already running processes. **Trin-Trin** does not operate any differently in this mode; however, the output call graphs may be different. For instance, the dynamic call graph generated by the Analytics Engine may contain multiple top-level nodes, may have missing thread-start-points. This can be attributed, in part, to (a) some events may have occurred before **Trin-Trin** was attached to the application and (b) for pair events, such as events corresponding to spawning of a child thread by a parent thread, **Trin-Trin** may encounter only one event of the pair. **Trin-Trin** attempts to detect such cases and extract best effort call graphs.

4 Analytics Engine

In this section, we present the Analytics Engine. The Analytics Engine is built using Ocaml [60] and uses the ocamlgraph package [40]. Figure 8 shows the Analytics Engine pipeline. The pipeline has five stages. We walk through each stage in detail in the rest of this section.

4.1 Per-Thread Call Graph

As discussed in Sect. 3.3, **Trin-Trin** captures the dynamic call metadata on a per-thread basis. This stage of the Analytics Engine pipeline reads this metadata and generates corresponding call graphs for each thread. Figure 9 exemplifies per-thread partial dynamic call graphs.

Next, we walk through the nomenclature of the annotation of nodes and edges using Fig. 9c.

- *Node Annotation* It consists of a two element vector, where the first element corresponds to the call count and the second element corresponds to the coverage of the function corresponding to the given node. The suffixes of the annotation vectors in the nodes corresponding to the functions `a` and `b` denotes the thread number. Hot nodes are colored in shades of red; the threshold for hotness can be configured by the user.
- *Edge Annotation* Except for coloring, edge annotation is similar to node annotation.

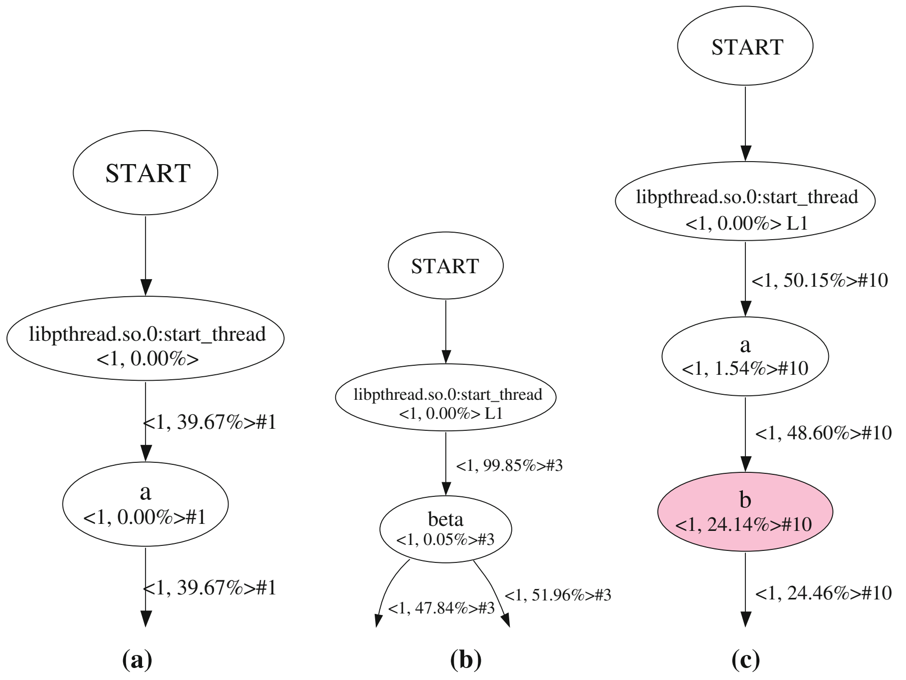


Fig. 9 Per-thread call graphs

A root node, called `START`, is introduced in each per-thread call graph. Except in Thread #0, `START` is translated, during the merging stage of the Analytics Engine (discussed in Sect. 4.2), to the function which spawned the given thread.

4.1.1 Handling Recursion

Let us revisit the example shown in Fig. 1. The call stack for the example is shown in Fig. 11a. Given the aforementioned assumptions, we note that the following three simple cycles exist (note that the first two form a complex cycle):

$$foo \rightarrow bar \rightarrow foo$$
$$foo \rightarrow toy \rightarrow bar \rightarrow foo$$
$$toy \rightarrow bar \rightarrow toy$$

There are two backedges: $bar \rightarrow foo$ and $bar \rightarrow toy$. The backedges (colored in purple) with their respective counts (annotated with `rc =`) are shown in the partial call graph obtained via **Trin-Trin** in Fig. 11b. Now let us consider the scenario wherein, using a sampling call graph extractor (such as `vtssrun`), samples are collected at levels 2 and 5. The two samples will be able to detect only one cycle ($foo \rightarrow bar \rightarrow foo$) in the call graph. In cases where a function is called in

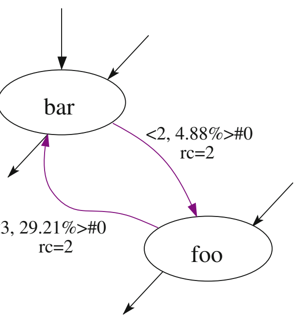


Fig. 10 An illustrative call graph

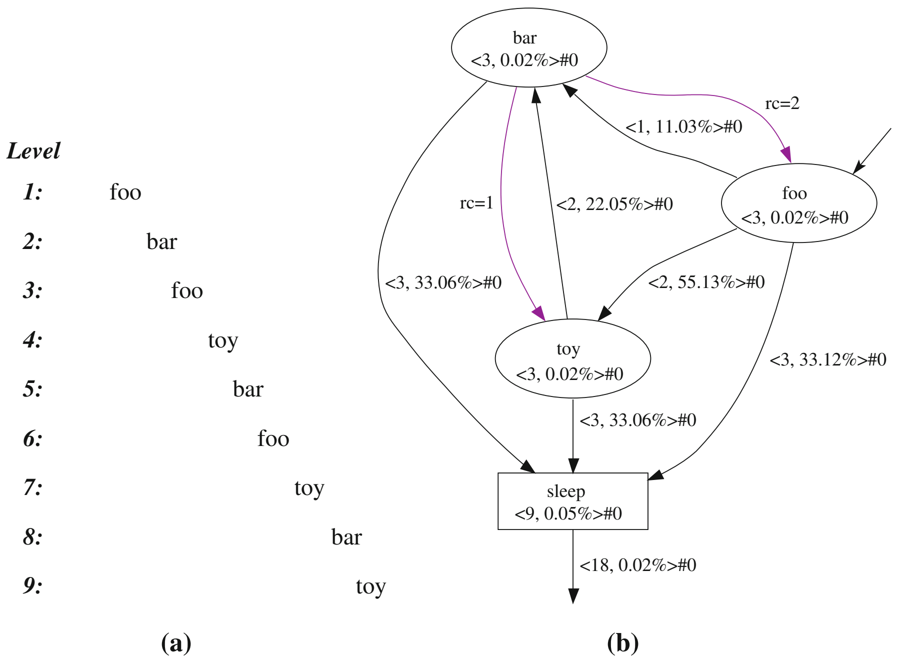


Fig. 11 Revisiting example shown in Fig. 2

both recursive and non-recursive contexts, sampling can potentially mislead program analysis.

For example, let us consider the partial call graph shown in Fig. 10. From the figure we note that the edges $foo \rightarrow bar$ and $bar \rightarrow foo$ serve as both normal (signified by the edge vector) and back-edges (signified by `rc =`). In such as scenario, sampling based approach may result in an higher call count corresponding to the edge vector and correspondingly result in a smaller count for `rc` (Fig. 11).

Fig. 12 **Trin-Trin** thread map file

```
A B          C
0 0x80484e8  1    A = Parent Thread Id.
0 0x80484e8  2    B = Parent Function
2 0x804848d  3    C = Child Thread Id.
0 0x804856c  4
0 0x80485ce  5
```

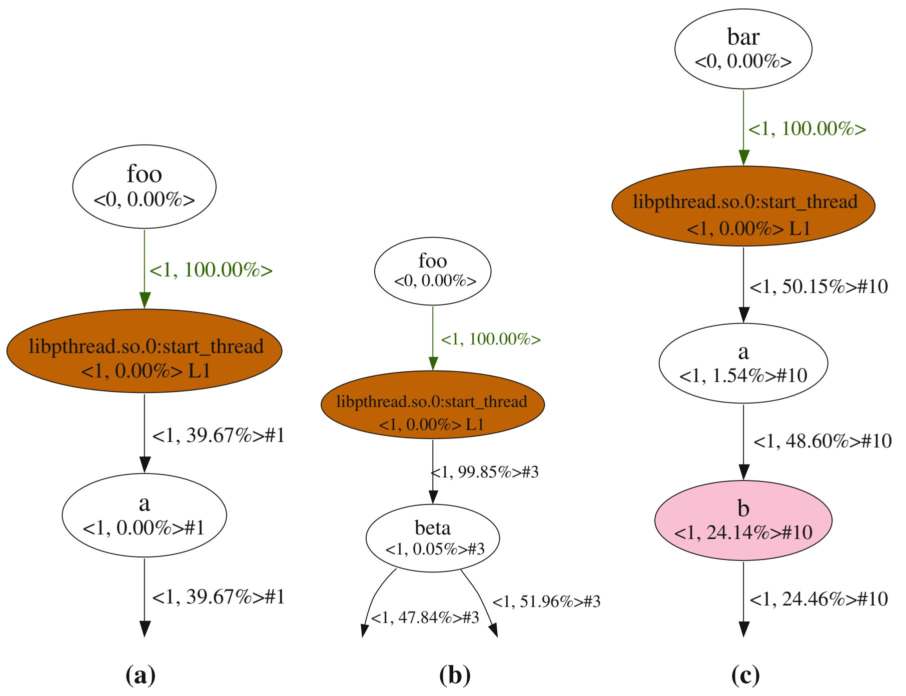


Fig. 13 Augmented per-thread call graphs

4.2 Merging

Trin-Trin generates thread map files on a per-process basis. A sample thread map file is shown in Fig. 12. Generated thread map files contains three fields for each spawned thread, viz., 'Parent Thread Id', 'Parent Function' and 'Child Thread Id'.

The Analytics Engine uses this information to merge the per-thread call graphs (generated in the first stage of the Analytics Engine, refer to Sect. 4.1).

Specifically,

- The `START` node of a per-thread call graph is translated, using the thread map information, to the function that spawned the thread under consideration, e.g., see the per-thread call graphs shown in Fig. 13 (and compare them with the per-thread call graphs shown in Fig. 9).
- Next, multiple invocations of a given function by different threads, if applicable, are merged. Merging all the invocations of the first function executed by the different threads (`start_thread` in Fig. 9) may induce spurious cycles in the call

graph. To avoid this, the set of nodes corresponding to the first function executed by the different threads is partitioned into subsets based on the nesting level of the different threads. The nodes in each subset are then merged.
In addition, the Analytics Engine can be configured to group threads based on their 'Parent Thread Id' or to avoid grouping, which will result in generation of unique spawn site node(s) for each thread.

The first function executed by any thread, except Thread #0, is colored in brown color to signify a thread spawning site in a composite dynamic call graph. Further, the creator of a thread and the first function executed by a thread are linked by a green edge (see Fig. 13). These green edges have an interesting property, they highlight the relative execution time of spawned thread(s) compared to the execution time of the application (aka., Thread #0). For e.g., in Fig. 14a the green edge connecting 'foo' to 'start_thread' shows that the six spawned threads consumed wall time equivalent to '35.36 %' of total execution time. Using 'Demand-Driven Context Extraction' (Sect. 4.4) this information can be obtained on a per thread basis as depicted in Fig. 14b.

Note that in the figure, a suffix is introduced to the annotation vector in the node corresponding to the function `start_thread`. The suffix denotes the nesting of multithreading.

The composite dynamic call graph corresponding to the per-thread call graphs shown in Fig. 13[2] is illustrated in Fig. 14a. Observe that annotation vector of the nodes corresponding to the functions `a` and `beta` has been extended to 3 elements. The third element of the vector represents *concurrency*, i.e., how many unique threads executed the function corresponding to the given node.

4.3 Pruning

This stage of the Analytics Engine facilitates pruning of nodes or edges with low coverage. In addition, pruning of low level functions, starting with '__' as illustrated in Fig. 2a, in shared libraries such as `libc` is also supported. The user of **Trin-Trin** can define the pruning filter by specifying a coverage threshold or via regular expressions. Pruning of "unimportant" nodes/edges augments the readability of a call graph. This is of particular importance in complex applications such as the Y! properties whose call graphs comprise of >10K nodes.

Filtering of nodes may necessitate the creation of *dfa* edges, as exemplified in Fig. 2. The different scenarios in which the pruning stage creates one or more *dfa* edges are enumerated below:

(i) Pruning of nodes may lead to orphaning of nodes in the resulting call graph. The orphaned nodes are then connected to their closest ancestor(s) via *dfa* edge(s).
(ii) Pruning of nodes may give rise to nodes with no child nodes. Such nodes are connected with their grand-child nodes (in the unpruned call graph), if any, via *dfa* edges.

[2] Six other per-thread call graphs were not shown in Fig. 13 for clarity purposes.

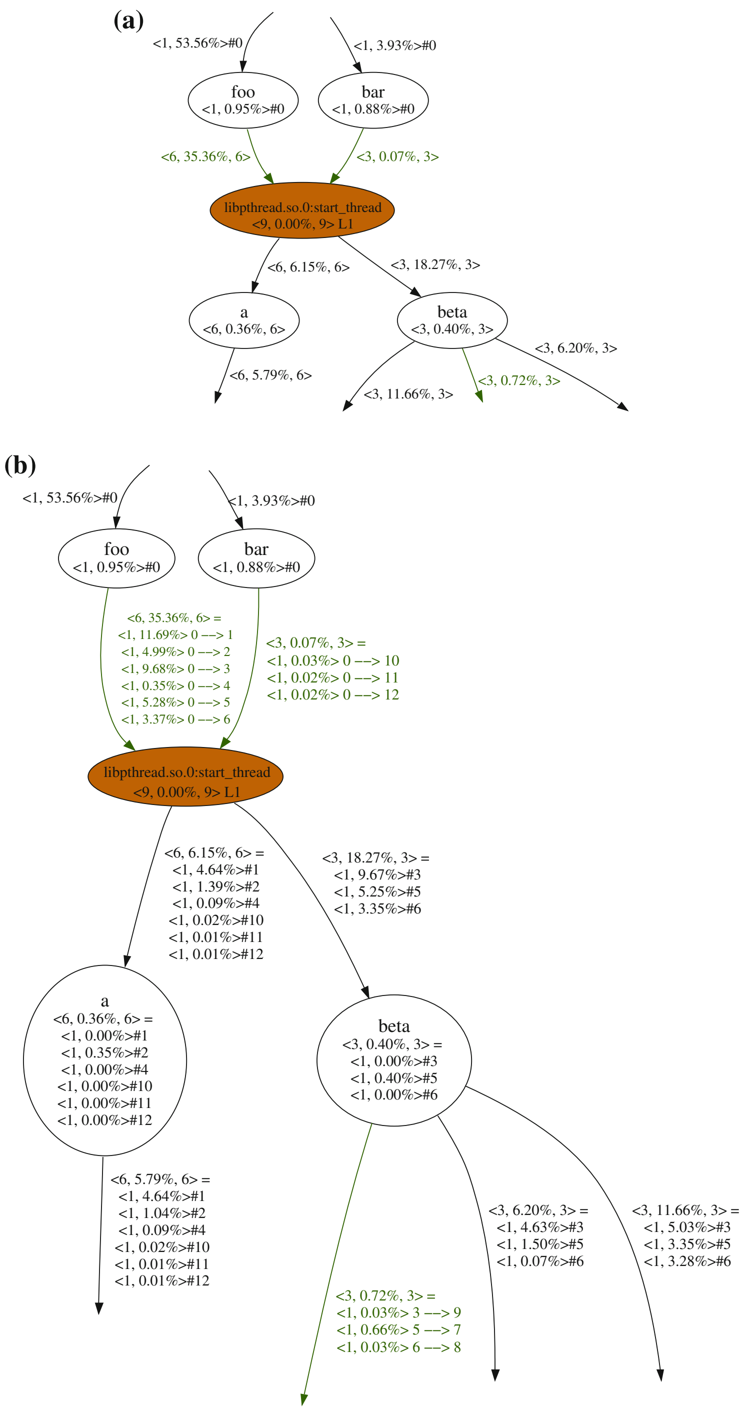


Fig. 14 Illustration of context extraction

4.4 Demand-Driven Context Extraction

Let us consider the partial call graph shown in Fig. 14a. We note the following from the figure: (a) functions `foo` and `bar` spawn 6 and 3 threads respectively and (b) the callees of `foo, bar` are the functions `a` (with a call count of 6) and `beta` (with a call count of 3). Using the partial call graph shown in Fig. 14a it is not possible to determine the caller–callee relationship between { `foo, bar`} and {`a, beta`} and the corresponding call counts. The above information can potentially gear optimization such as inlining and function versioning.

To this end, the Analytics Engine facilitates demand-driven context extraction. For example, Fig. 14b details the thread context for the partial call graph shown in Fig. 14a. Note that vector corresponding to the edge $foo \rightarrow libpthread.so.0 : start\_thread$ is dissected into 6 thread-specific vectors. The notation `0 −> 1` signifies that Thread #0 spawned Thread #1.

From the thread-specific vector inside the node corresponding to function a, we note that Thread #1 executed function `a`. On further analysis, based on the context information, one can conclude that the function `beta` was called *only* by function `foo`, whereas the function `a` was called by both the functions `foo` and `bar`.

4.5 Summary Generator

In this stage, the Analytics Engine generates a summary of the dynamic call graph extracted at run time. The summary comprises of metrics which characterizes the dynamic call graph such as number of nodes, number of edges in the dynamic call graph. Additionally, program properties such as number of direct and indirect calls, hottest function, hottest edge, are reported. Additionally, the shared libraries used by the application and their respective coverages are also reported. Summary for the example shown in Fig. 2, obtained via **Trin-Trin**, is given in Fig. 15.

4.6 Discussion

In this subsection we discuss other salient features of the Analytics Engine which assist program analysis and can potentially guide program optimization.

Fig. 15 **Trin-Trin**'s summary for the example shown in Fig. 2

```
           Node Count: 9
           Edge Count: 13
         Normal Edges: 10
            DFA Edges: 3
         Thread Edges: 0
 Multi-threaded App.: No
          Cycle Count: 3
         Direct Calls: 36
Hot Node Information: [sleep] <9, 0.05%>#0
Hot Edge Information: main->foo <1, 99.30%>#0
      Used Libraries: libc.so.6, 99.44%
                      ld-linux-x86-64.so.2, 0.12%
                      Example, 0.18%
                      libpthread.so.0, 0.25%
```

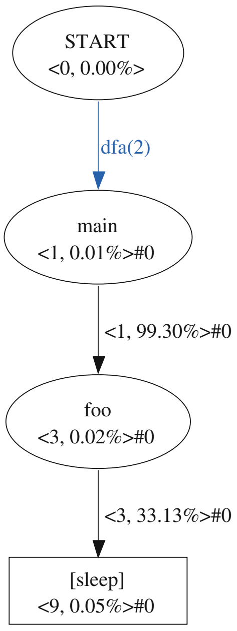


Fig. 16 Edge-profiled call graph for the example shown in Fig. 1

4.6.1 Hot Node and Hot Path Analysis

As discussed in Sect. 4.3, the Analytics Engine generates 'Node-profiled' call graph, i.e., it ranks nodes based on their coverages and then prunes the call graph using a 'user-defined' threshold. This combined with node coloring (as explained in Sect. 9) enables the users of **Trin-Trin** to quickly drill-down on functions with the most bang-for-the-buck from optimization standpoint.

The Analytics Engine also facilitates extraction of 'Edge-profiled' call graphs. In this mode, during the *Pruning* stage, the nodes are ranked based on the coverage of connecting edges. Edge-profiled call graphs highlight the hot paths in a given application. Using a threshold of 10 %, the edge-profiled call graph for the example in Fig. 1 is shown in Fig. 16. On comparing Figs. 11b and 16, we note that the latter corresponds to the hottest path in the former!

4.6.2 Run Time Analysis

In client-server applications such as the Y! properties, threads may be idle for various reasons such as, but not limited to, waiting to acquire a lock, I/O. In order to analyze the run time behavior, it is important to quantify the amount of time being spent in functions such as `epoll_wait`, `pthread_cond_wait`, `pthread_cond_timedwait`, `sem_wait`, `___lll_mutex_lock_wait` and `pread`. This is not possible when using tools such as `vtssrun` as they capture CPU cycles in the user space. In contrast, **Trin-Trin** reports the total wall clock time spent in such functions. We have successfully used **Trin-Trin** to quantify the wall clock time spent in boost and pthread-based locking routines in Y! seach engine and other Y! properties.

5 Experiments

In this section, we present an evaluation of **Trin-Trin**'s run time overhead of **Trin-Trin** using several Unix utilities, applications from the industry-standard SPEC CINT2006, CFP2006 benchmark suite. Additionally, we illustrate, using real programs, how **Trin-Trin** can be used to guide program optimization.

5.1 Setup

Table 1 lists the open-source applications and Yahoo! internal applications used in the experiments. The configuration of the system used for running the applications listed in Table 1 is given in Table 2.

5.2 Overhead and Dynamic Call Count Analysis

We dissect the run time overhead incurred, when running an application under the Pin environment with **Trin-Trin**, into the following three categories:

- *Standalone Pin overhead* (O_p): Measures the analysis and runtime overhead introduced by the Pin environment. We measured this overhead by building a replica of the **Trin-Trin** Pin tool that contained empty analysis routines. As a consequence no instrumentation was insterted at runtime.
- *Pin tracing overhead* (O_t): Measures the minimum analysis and runtime overhead induced by Pin for a given application. We measured this overhead by tracing applications using an empty 'Shared Library and Routine Filter' (refer to Sect. 3.1), causing **Trin-Trin** to not instrument any routines at runtime.
- **Trin-Trin** *overhead* ($O_{\mathbf{TT}}$): Measures the **Trin-Trin** overhead when tracing all dynamic calls in an application.
- **Trin-Trin** *with Min–Max overhead* ($O_{\mathbf{TTMM}}$): Measures the run time overhead introduced by enabling capture of minimum and maximum call coverage for each instrumented edge. For this, conditionals expressions need to be added to **Trin-Trin**'s default Pin instrumentation routines.

The aforementioned overheads for 64-bit SPEC CINT2006 and CFP2006 applications are shown in Fig. 17a. Excluding the outlier `483.xalancbmk`, from the figure we observe that the Pin environment introduces an average overhead of 29 %. Pin-based tracing introduces an average overhead of 12 % and **Trin-Trin** introduces an overhead on 141 % on an average. Also, from Fig. 17a we note that O_{TT} varies significantly (standard deviation of 179 %) across the different applications. To reason this, we obtained the total dynamic call counts via **Trin-Trin**. Excluding the outlier, we regressed O_{TT} with respect to the total dynamic call count, see Fig. 18. From the figure we conclude that the applications with high call count have high O_{TT}.

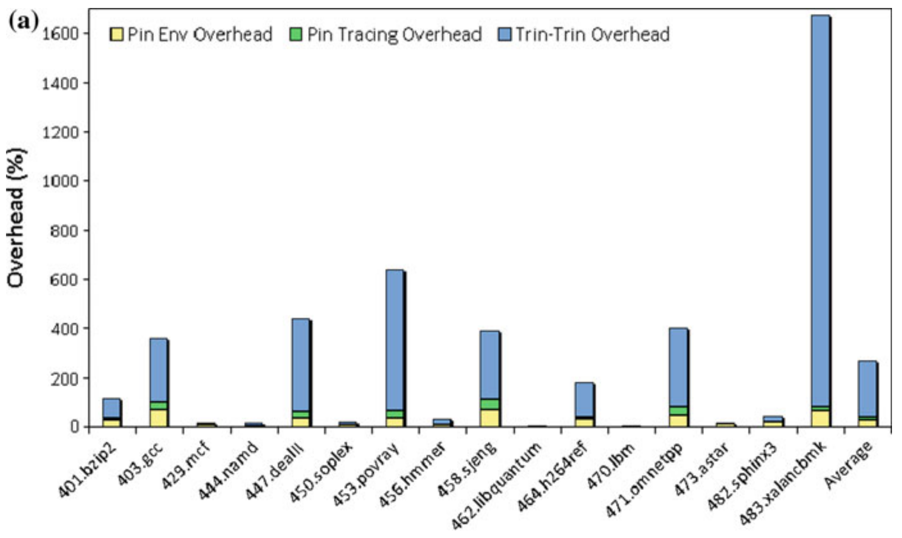
Table 1 Application suite

Benchmark	# of instructions
Open source utilities	
find (4.1.20)	10,412
gzip(1.3.3)	10,050
tar (1.14)	30,855
dd (5.21)	4,085
SPEC CINT2006	
401.bzip2	123,628
403.gcc	1,085,855
429.mcf	99,281
445.gobmk	459,579
456.hmmer	171,340
458.sjeng	132,535
462.libquantum	149,719
464.h264avc	310,614
471.omnetpp	449,153
473.astar	132,218
483.xalancbmk	1,261,823
SPEC CFP2006	
444.namd	198,937
447.dealII	894,761
450.soplex	317,300
453.povray	472,821
470.lbm	98,651
482.sphinx3	177,187
Y!	
Y! search engine	$>4M$

Table 2 Experimental setup

Processor	Intel Xeon®CPU E5530, 2.40 GHz
Memory	2 GB
L1 D-cache	32 KB
L1 I-cache	32 KB
L2 cache	256 KB
L3 cache	8 MB
Intel QPI speed	5.86 GT/s
Compiler and flags	icc (v 11.1)-fast
OS	Linux 2.6.9-80.ELlargesmp #1 SMP

Table 3 lists the Pearson's correlation [38,47] between overhead and dynamic call count. From the table we note that each type of overhead has a high positive correlation coefficient which implies that the overhead increases with an increase in dynamic call count.



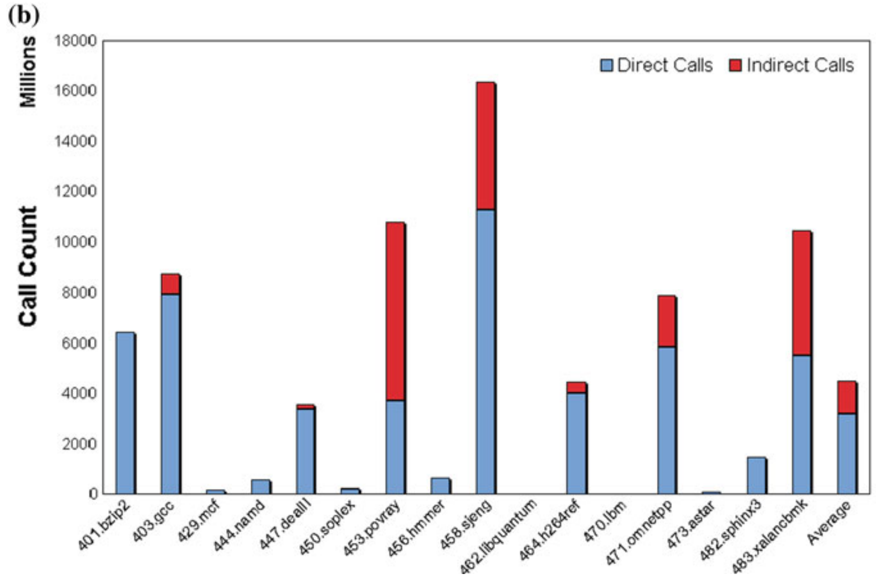


Fig. 17 **a** Run time overhead, **b** dynamic call counts

High value of O_{TT} in applications such as `403.gcc` can, in part, be attributed to the flat function coverage profile (see Fig. 4). In such cases, the overhead incurred to profile a function may outweight the self-coverage of the function itself. In contrast, applications such as `470.lbm` incur $\sim 1\,\%$ overhead as a single function in the application has a coverage of >95 %.

Excluding the outlier `483.xalancbmk`, we note that a higher O_p, O_t corresponds to a higher value of O_{TT}. This evidenced from the overhead correlation matrix shown

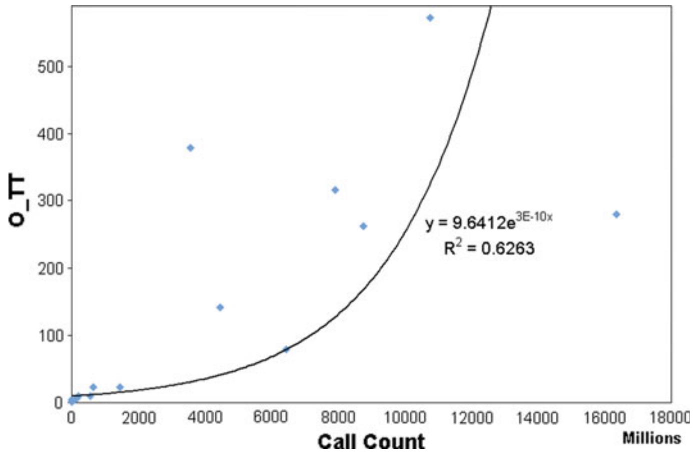


Fig. 18 Regression between dynamic call count and total overhead

Table 3 Correlation between overhead and dynamic call count

	O_p	O_t	O_{TT}
Including `483.xalancbmk`	0.90	0.87	0.59
Excluding `483.xalancbmk`	0.89	0.88	0.76

Table 4 Overhead correlation matrix

	O_p	O_t	O_{TT}
O_p	x	0.878	0.7
O_t		x	0.906
O_{TT}			x

in Table 4. For instance, the correlation coefficient between O_t and O_{TT} is 0.906 which implies that the two metrics are highly correlated.

The dissection of the total dynamic call count into direct calls and indirect calls is shown in Fig. 17b. The dynamic call count captured by **Trin-Trin**, on a per function basis, has twofold benefits:

(i) It helps to determine the coverage per call of a given function. This is of paramount importance when making the decision whether to optimize a given function or to address minimization of calls to the function. The latter is preferred when coverage per call is very low.

(ii) The dissection of call count into direct and indirect calls can also guide program optimization. For example, a high percentage of indirect calls, as in `483.xalancbmk` (see Fig. 17b), suggests the conversion of virtual functions to concrete functions may yield better performance.

For the open source applications `find`, `gzip` and `tar` we used the FreeBSD 8.1 Ports Distribution (available from [17]) as the input. Specifically, we extracted the ports tar-gz file in 5 different directories and used these directories as input to *tar* and

Table 5 Overhead for Linux utilities

	O_p	O_t	O_{TT}
find	14.89	0.95	6.44
gzip	14.60	7.19	113.85
tar	17.79	2.92	3.31
dd	0.74	0.01	0.03

find applications (viz. `'tar -cf ports.tar ports ports-2 ports-3 ports-4 ports-5'` and `'find ports ports-2 ports-3 ports-4 ports-5 -name "nonexistent"'`). For the *gzip* test we concatenated the decompressed ports tar file five times to create the input data set and benchmarked *gzip* execution by compressing this data set. We benchmarked `dd` using the following command:

```
/bin/ddif = /dev/sdbof = /dev/null bs = 65536 count = 409600
```

The overhead observed for these utilities is tabulated in Table 5. Note that the total overhead observed in the case of `dd` was less than 1 %.

In light of the fact that Pin is 3.3× faster than `valgrind` (showed by Luk et al. in [31]) and the fact that `Callgrind` is based on `valgrind`, the overhead incurred by `Trin-Trin` is substantially lower than `Callgrind`.

5.2.1 Trade-Off Between Overhead and Precision

As mentioned earlier, **Trin-Trin** enables capture of minimum and maximum call durations for each instrumented edge. This induces additional overhead, as tabulated in Table 6. The results in the table suggest that the addition of trivial calculations such as min and max can cause a significant overhead. For example, ($O_{\mathbf{TTMM}}$) for `401.bzip2` is approximately 6 % higher than the ($O_{\mathbf{TT}}$).

The coverage of a function may vary across different calls owing to a different control flow in the different calls. **Trin-Trin** facilitates capturing of an outlier. In case there exists an outlier, the corresponding function can be versioned [18]. This can in turn potentially guide function inlining if the coverage of the calls other than the outlier is low.

One common feedback from various teams at Yahoo! was to evaluate the impact of **Trin-Trin**'s overhead on the ordering of hot functions (a skew in the ordering of the hot functions would misguide the performance optimization efforts). For example, the Hadoop team [25] queried the impact of the overhead of **Trin-Trin** on the ordering of hotness of `bzip2` which is used for compressing output streams [44] and de-compressing input streams [43]. To this end, we computed the Spearman's correlation [52,53] coefficient (ρ) between the top 15 hot functions reported by **Trin-Trin** and Intel's PTU [26] for `401.bzip2` [57], see Table 7, using the five data sets of the reference input.

Table 6 Trade-off between overhead and precision

Benchmark	Overhead (%)	
	Without Min–Max ($O_{\mathbf{TT}}$)	With Min–Max ($O_{\mathbf{TTMM}}$)
401.bzip2	114	121
403.gcc	362	368
429.mcf	16	16
444.namd	16	16
447.dealII	443	460
450.soplex	19	19
453.povray	640	667
456.hmmer	31	33
458.sjeng	389	400
462.libquantum	3	3
464.h264ref	182	187
470.lbm	6	6
471.omnetpp	400	405
473.astar	14	45
482.sphinx3	44	45
483.xalancbmk	1,671	1,680

Table 7 Spearman's correlation for `401.bzip2` [57]

	Spearman's ρ
401.bzip2-1	0.927
401.bzip2-2	0.938
401.bzip2-3	0.862
401.bzip2-4	0.924
401.bzip2-5	0.928

The high value of Spearman's ρ highlights that **Trin-Trin**'s overhead does not skew the ordering of the hot functions in a significant fashion.

5.3 Case Study: Identifying Peformance Bottlenecks

As mentioned earlier, function coverage reported by existing tools such as `Callgrind` [8] is based on CPU cycles instead of the wall clock time/run time. This limits identification of performance bottlenecks in applications such as MySQL [37] where disk access account for a large percentage of the run time. The following describes how **Trin-Trin** was used to diagnose a performance bottleneck in one of the Yahoo! properties which uses MySQL at the back end.

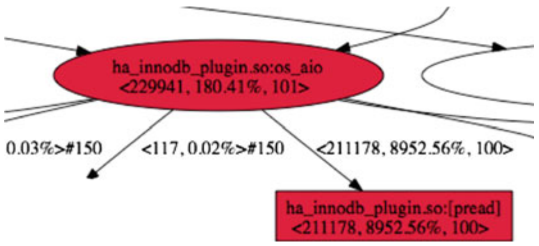


Fig. 19 Partial call graph, obtained via **Trin-Trin**, highlighting the hottest function in MySQL

Table 8 Reduction in 95th and 99th percentiles for read and write latencies

% Reduction			
Read latency		Write latency	
95th percentile	99th percentile	95th percentile	99th percentile
611 %	66 %	105 %	47 %

Table 9 Reduction in 95th and 99th percentiles for `GET`s

% Reduction	
95th percentile	99th percentile
31 %	191 %

We profiled MySQL using **Trin-Trin**. A partial graph highlighting the hottest function is shown in Fig. 19. From the call graph we learned that heavy disk I/O was the primary cause of high latency of the Yahoo! property. This guided the team to address optimization of InnoDB disk I/O [42].

In addition, to mitigate the impact of disk I/O on latency, we tuned the Apache configuration. Specifically, we decreased the number of Apache worker processes and increased the value of the `backlog` parameter so as to maintain the simultaneous connection handling capacity. The decrease in 95th and 99th percentiles for read and write latencies are reported in Table 8.

Lastly, the performance of `GET`s for the Yahoo! property was improved by extending the MySQL queries using a `where` clause. The performance improvements for gets is reported in Table 9.

5.4 Case Study: Peformance Regression Analysis

In a production setting, it is not uncommon that 32-bit binaries are run on 64-bit systems. The switch to a 64-bit binary can potentially result in a performance regression. Table 10 lists the regression we observed for four applications in SPEC CPU2006 when switching from a 32-bit binary to a 64-bit binary.

Trin-Trin can assist to diagnose the above with respect to the impact on dynamic call counts. Figure 20 shows the difference in the total dynamic call count between the 32- and 64-bit runs. Observe that the total dynamic call count increased for four

Table 10 Performance degradation when using 64-bit binary

	% Degradation
403.gcc	22.93
429.mcf	14.05
450.soplex	1.45
471.omnetpp	15.29

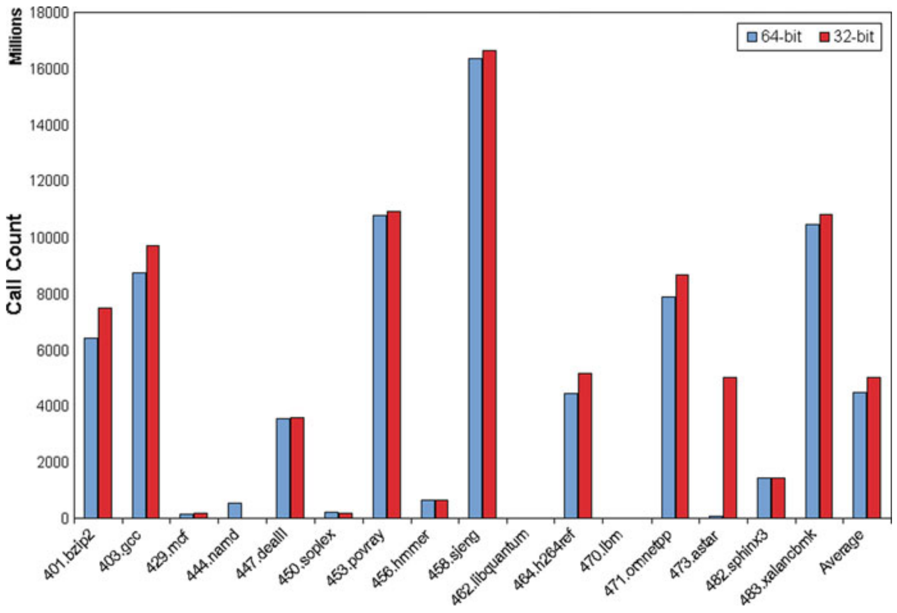
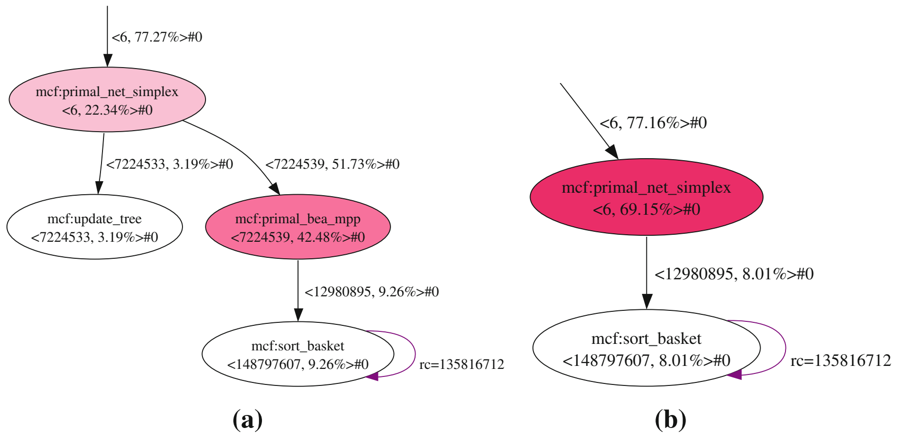


Fig. 20 Dynamic call count of 32-bit and 64-bit binaries

applications, such as `450.soplex`, in the 64-bit case. This can potentially guide a compiler writer and/or a performance enginner to tune the cost model which drives function inlining. For example, Fig. 21a, b, obtained via **Trin-Trin**, illustrate the difference in inlining of the functions `primal_bea_mpp` and `update_tree` in 32-bit and 64-bit cases. Note that the call count of the incoming edge to the node corresponding to the function `sort_basket` is consistent in both the cases.

It is well known that aggressive inlining can potentially result in performance degradation. In order to address whether aggressive inlining is the cause for the performance degradation in the 64-bit case for `429.mcf` (reported in Table 10), we disabled the inlining of the function `primal_bea_mpp` via `__attribute__((noinline))`. Figure 22b shows the partial call graph obtained subsequently. We observed that disabling inlining as mentioned above did not alleviate the performance degradation with respect to the 32-bit case.

To analyze the performance degradation further, we turned on the feature of **Trin-Trin** whereby the absolute time is included in the node and edge annotation vectors. From Fig. 22a, b, we note that the time spent in `primal_bea_mpp` in the 32-bit and 64-bit cases are in the same ballpark; on the other hand, the time spent in



<6, 77.27%>#0
mcf:primal_net_simplex <6, 22.34%>#0
<7224533, 3.19%>#0
<7224539, 51.73%>#0
mcf:update_tree <7224533, 3.19%>#0
mcf:primal_bea_mpp <7224539, 42.48%>#0
<12980895, 9.26%>#0
mcf:sort_basket <148797607, 9.26%>#0
rc=135816712
(a)
<6, 77.16%>#0
mcf:primal_net_simplex <6, 69.15%>#0
<12980895, 8.01%>#0
mcf:sort_basket <148797607, 8.01%>#0
rc=135816712
(b)

Fig. 21 Difference in inlining in **a** 32-bit and **b** 64-bit binaries

Table 11 Performance degradation of hottest functions in `429.mcf`

Compiler	% Degradation: 64-bit w.r.t. 32-bit	
	`Primal_bea_mpp`	`Refresh_potential`
icc (no ipo)	18.74	90.39
gcc (-O3)	16.78	99.51

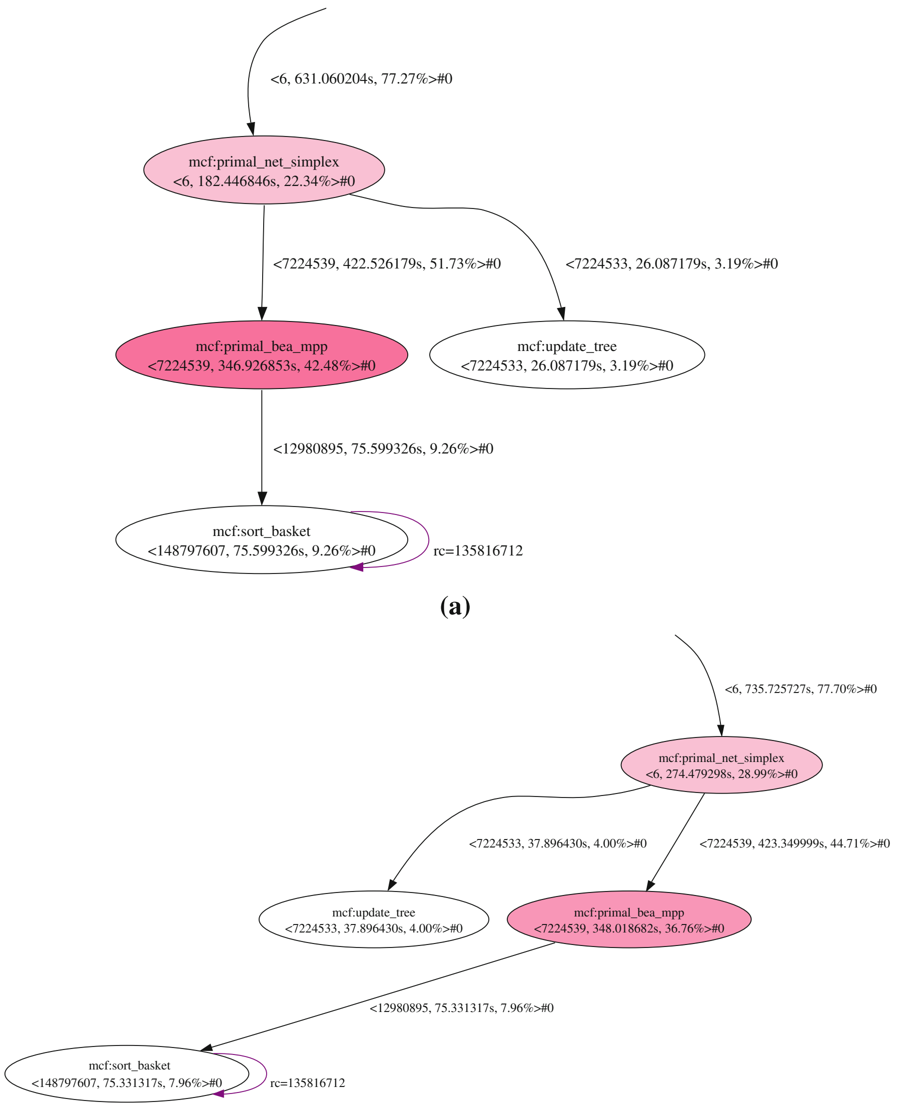
`primal_net_simplex` is 50.4 % higher in the 64-bit case as compared to the 32-bit case. We observed a similar performance degradation when gcc v.3.4.6 was used to generate the 32- and 64-bit binaries. Using **Trin-Trin**, we identified the primary sources of the performance degradation for the top two hottest functions (highlighted in shades of red by of **Trin-Trin**) `429.mcf`. Table 11 reports the percentage degradation when `429.mcf` was compiled with Intel's *icc* (with interprocedural optimization (ipo) disabled) and *gcc*.

Given that the partial call graphs shown in Fig. 22 have the same structure with respect to caller–callee relationships, the performance degradation is indicative of inefficient code generation in the 64-bit case.

As another case study, we used **Trin-Trin** to reason the performance regression (=12 %) observed with the 32-bit binary of `473.astar`. Analysis of the call graphs, extracted via **Trin-Trin**, corresponding to the 32-bit and 64-bit binaries highlighted that the hot function `releasepoint` was not inlined in the 32-bit case. In light of this, we inlined the function `releasepoint` which alleviated the performance regression mentioned above.

6 Previous Work

In this section, we present an overview of related work. Several tools such as `cscope` [12] and `doxygen` [14] are available for code browsing and analysis. For example, `cscope` can be used to determine the caller–callee caller–callee relationship



<6, 631.060204s, 77.27%>#0
mcf:primal_net_simplex
<6, 182.446846s, 22.34%>#0
<7224539, 422.526179s, 51.73%>#0
<7224533, 26.087179s, 3.19%>#0
mcf:primal_bea_mpp
<7224539, 346.926853s, 42.48%>#0
mcf:update_tree
<7224533, 26.087179s, 3.19%>#0
<12980895, 75.599326s, 9.26%>#0
mcf:sort_basket
<148797607, 75.599326s, 9.26%>#0
rc=135816712
(a)
<6, 735.725727s, 77.70%>#0
mcf:primal_net_simplex
<6, 274.479298s, 28.99%>#0
<7224533, 37.896430s, 4.00%>#0
<7224539, 423.349999s, 44.71%>#0
mcf:update_tree
<7224533, 37.896430s, 4.00%>#0
mcf:primal_bea_mpp
<7224539, 348.018682s, 36.76%>#0
<12980895, 75.331317s, 7.96%>#0
mcf:sort_basket
<148797607, 75.331317s, 7.96%>#0
rc=135816712
(b)

Fig. 22 Comparing absolute times. **a** 32-bit, **b** 64-bit

between the different functions in a C code base. The relationships are established pre-compile time and hence `cscope` cannot detect the dynamic (e.g., in presence of virtual function calls) caller–callee relationships. In [13], Demme and Sethumadhavan highlighted the importance of precise profiling, using hardware peformance counters, on analyzing production software such as MySQL [37] and Firefox [16].

In [54], Spinellis presented a tool, `CScout`, for source code analysis of C programs. A web-based user interface is provided in the front-end for source code navigation and an sql-based back-end is provided for more complex source code analysis and manipulation. Recent work from Tallent and Mellor-Crummey [59] is centered around assessing the extent of parallel idleness and parallel overhead in multithreaded applications. On the other hand, as discussed earlier in the paper, **Trin-Trin** exposes the context information to guide optimization of multithreaded applications. Thus, **Trin-Trin** is complementary to the work presented by Tallent and Mellor-Crummey.

6.1 Static Call Graph Extractors

Several static call graph extractors have been proposed in the past. For example, `rigiparse` [35], `CIA` [9], `cawk` [21], `Field` [48] and `cflow`—this tool is distributed with some UNIX systems.

In [36], Murphy et al. presented an empirical study of nine static call graph extractors. Their qualitative analysis shows that there is large variation in the call graphs extracted using different static call graph extractors. The variation can, in part, be ascribed to differing treatments of macros, function pointers, input formats et cetera.

Milanova et al. [33] presented an approach to generate precise call graphs in the presence of function pointers. Unlike dynamic call graphs, the precision of static call graphs is subject to the precision of the pointer analysis algorithm used for disambiguation. Milanova et al. used a flow- and context-insensitive pointer analysis algorithm; hence, the calling context of functions invoked in a multithreaded scenario cannot be captured using their approach.

Tip and Palsberg proposed scalable propagation-based call graph construction algorithms in [61]. They investigate the design space between the RTA [4] and 0-CFA [51] algorithms[3] while constructing propagation-based call graphs. In [22], Grove and Chambers presented a framework for understanding call graph construction algorithms. Cross-algorithm performance comparisons can also be performed using the framework. Unlike **Trin-Trin**, the call graphs are generated at compile time in Grove and Chambers' framework. Recently, Zhang and Ryder [64] explored approaches for generating application call graphs for Java. They proposed a data reachability algorithm to resolve library callbacks accurately and showed that the algorithm reports fewer number of spurious callback edges in a static call graph. Lhotak [30] presented a tool for comparing call graphs. The call graph difference search tool ranks the call graph edges by their likelihood of causing large differences in the call graphs.

6.2 Production/Open Source Tools

Graham et al. [19] presented `Gprof`—a profiler which reports the running time of called routines and the running time of the routines that call them. Unlike **Trin-Trin**, `gprof` requires the application to be compiled with `-pg` option. This is typically not feasible in a production scenario. Further, akin to `vtssrun`, `gprof` does not support call graph extraction of already running applications. Eustace and Srivastava

[3] These algorithms are used to approximate run-time values of expressions. A key property of these algorithms is that they do not analyze values on the run-time stack.

[15] presented a tool for code instrumentation, called `ATOM`. Unlike Pin, `ATOM` is not capable of dynamically injecting instrumentation into a running executable.

`vtssrun` [27] is a tool (from Intel) for statistical call graph data collection. Specifically, it employs statistical sampling to capture the call stack. The latest version of `vtssrun` (version 3.2, Update 1, build 8240 of the Intel Performance Tuning Utility) has the following limitations:

- The target application is required to be run under the `vtssrun` envelope. This is severely limiting in a production scenario. To this end, **Trin-Trin** enables extraction of dynamic call graph of an already running application. **Trin-Trin** uses Pin's APIs to attach and detach to/from an already running process [31].
- `vtssrun` lacks analytics support for program analysis.
- It does not provide a graphical interface to view a dynamic call graph.

`Oprofile` [41] supports call graph profiling similar to `vtssrun` and has similar limitations.

The framework presented in this paper is similar to the tool `Callgrind` [8]. `Callgrind` is based on `valgrind` [62]. The call graph profile data dumped by `Callgrind` can be viewed graphically using `KCachegrind` [28]. In [6], Bruening presented a dynamic instrumentation tool `DynamoRIO`. Unlike `valgrind` and `DynamoRIO`, Pin supports register allocation, inlining, liveness analysis and instruction scheduling to optimize jitted code.

6.3 Pin-based Tools

Several Pin-based program analysis tools have been proposed. In this section we overview these tools and highlight how they differ from **Trin-Trin**. In [34], Moseley et al. presented an approach for loop-centric profiling. The profiling is done using a Pin-based tool. A hierarchical view of how much time is spent in a given loop and the loops nested within it is provided. Stube et al. [55] provide Pin-based software probes for machine characterization and application performance prediction. Patil et al. [46] present a Pin-based framework—PinPlay—for deterministic replay and reproducible analysis of parallel programs. In [3], Bach et al. detail how Pin can be used to analyze parallel programs. None of the aforementioned Pin-based tools are geared towards the extraction of dynamic call graphs. From a program analysis standpoint, we believe **Trin-Trin** is complementary to the existing Pin-based tools.

7 Conclusion

In this paper, we presented a Pin-based framework called **Trin-Trin** for extraction of complete, precise and dynamic call graphs. **Trin-Trin** can be used not only for sequential, but also for multithreaded and multi-process applications. A key highlight of **Trin-Trin** is that it can be used to extract dynamic call graphs of already running applications. This is of particular importance as applications in production cannot be restarted. The framework includes an analytics engine to assist a developer and/or a performance engineer. The analytics engine can be used to determine, for example, hottest path in a call graph, existence of cycles in a call graph, depth of recursion, levels of multithreading, enables demand-driven context extraction et cetera. In addition, the analytics engine supports graphical visualization of dynamic call graphs wherein nodes and edges are annotated with call metadata. The analytics engine is run post-collection of the call graph profile data; hence, the analytic engine does *not* introduce any run time overhead. We quantified the run time overhead incurred due to the Pin environment, due to Pin tracing and the overhead induced by **Trin-Trin**. Lastly, we presented a case study to illustrate how **Trin-Trin** can be used to analyze performance regressions. We have been using **Trin-Trin** for analysis of the run time performance of multiple Yahoo! properties such as the Y! the advertising platform. Recently, we extended support for extraction of dynamic call-graphs of Java based applications. Thus, use of **Trin-Trin** is not limited to C/C++ applications (unlike the tools mentioned in Sect. 6).

As future work, we plan to provide sampling support, configurable on a per-function basis, in **Trin-Trin**. In particular, the sampling mode would be set on-the-fly in order depending on, say, the hotness of the function. Another area of exploration will be Out-of-Band profiling. In this, we plan to limit the capture of metadata by Pin-analysis routines. Specifically, the run time information will be relegated to a separate process which would then generate 'Call Metadata' in its own context thereby reducing overhead in the application.

Lastly, we plan to investigate, using **Trin-Trin**, performance bottlenecks owing to synchronization placement [39] and memory bottlenecks [23,45] in open source and production software.

References

1. Allen, F.E.: Program optimization. Annu. Rev. Autom. Program. **5**, 239–307 (1969)
2. Allen, F.E.: Interprocedural data flow analysis. In: IFIP Congress, pp. 398–402 (1974)
3. Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.K., Lyons, G., Patil, H., Tal, A.: Analyzing parallel programs with Pin. Computer **43**, 34–41 (2010)
4. Bacon, D.F., Sweeney, P.F.: Fast static analysis of c++ virtual function calls. In: Proceedings of the 11th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, San Jose, CA, pp. 324–341 (1996)
5. Banning, J.P.: An efficient way to find the side effects of procedure calls and the aliases of variables. In: Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages, New York, NY, pp. 29–41 (1979)
6. Bruening, D.: Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, Massachusetts Institute of Technology (2004)
7. Callahan, D., Carle, A., Hall, M.W., Kennedy, K.: Constructing the procedure call multigraph. IEEE Trans. Soft. Eng. **16**(4), 483–487 (1990)
8. Callgrind: a call-graph generating cache profiler. http://valgrind.org/docs/manual/cl-manual.html
9. Chen, Y.F., Nishimoto, M.Y., Ramamoorthy, C.V.: The C information abstraction system. IEEE Trans. Softw. Eng. **16**(3), 325–334 (1990)
10. Chikofsky, E.J., Cross, J.H. II.: Reverse engineering and design recovery: a taxonomy. IEEE Softw. **7**(1), 13–17 (1990)
11. Choi, S.C., Scacchi, W.: Extracting and restructuring the design of large systems. IEEE Softw. **7**(1), 66–71 (1990)
12. Cscope: a developer's tool for browsing source code. http://cscope.sourceforge.net/

13. Demme, J., Sethumadhavan, S.: Rapid identification of architectural bottlenecks via precise event counting. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, pp. 353–364 (2011)
14. Doxygen. http://doxygen.org/
15. Eustace, A., Srivastava, A.: ATOM: a flexible interface for building high performance program analysis tools. In: Proceedings of the USENIX 1995 Technical Conference, New Orleans, LA, pp. 25–25 (1995)
16. Firefox. http://www.mozilla.org/en-US/firefox/new/
17. FreeBSD 8.1 ports distribution. ftp://ftp.freebsd.org/pub/FreeBSD/releases/i386/8.1-RELEASE/ports/. MD5:73589e78c9e246f737e43b8c57c8f875
18. Gerber, R., Bik, A.J., Smith, K.B., Tian, X.: The Software Optimization Cookbook. Intel Press, Hillsboro, OR (2006)
19. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. In: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, Boston, MA, pp. 120–126 (1982)
20. Graphviz. http://www.graphviz.org/
21. Griswold, W.G., Atkinson, D.C., McCurdy, C.: Fast, flexible syntactic pattern matching and processing. In: Proceedings of the 4th International Workshop on Program Comprehension, p. 144 (1996)
22. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst. **23**(6), 685–746 (2001)
23. Grun, P., Dutt, N., Nicolau, A.: Memory aware compilation through accurate timing extraction. In: Proceedings of the 37th Annual Design Automation Conference, Los Angeles, CA, USA, pp. 316–321 (2000)
24. Hall, M.W., Kennedy, K.: Efficient call graph analysis. ACM Lett. Programm. Lang. Syst. **1**(3), 227–242 (1992)
25. http://hadoop.apache.org/
26. Intel® Performance tuning utility 4.0 update 5. http://software.intel.com/en-us/articles/intel-performance-tuning-utility/
27. Intel® VTune. http://software.intel.com/en-us/intel-vtune/
28. KCachegrind, http://kcachegrind.sourceforge.net/html/Home.html
29. Lakhotia, A.: Constructing call multigraphs using dependence graphs. In: Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages, Charleston, SC, pp. 273–284 (1993)
30. Lhoták, O.: Comparing call graphs. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, San Diego, CA, pp. 37–42 (2007)
31. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the SIGPLAN '05 Conference on Programming Language Design and Implementation, Chicago, IL, USA, pp. 190–200 (2005)
32. McKeeman, W.M.: Peephole optimization. Commun. ACM **8**(7), 443–444 (1965)
33. Milanova, A., Rountev, A., Ryder, B.G.: Precise call graphs for C programs with function pointers. Autom. Softw. Eng. **11**(1), 7–26 (2004)
34. Moseley, T., Connors, D.A., Grunwald, D., Peri, R.: Identifying potential parallelism via loop-centric profiling. In: Proceedings of the 4th International Conference on Computing Frontiers, Ischia, Italy, pp. 143–152 (2007)
35. Müller, H.A., Klashinsky, K.: Rigi-a system for programming-in-the-large. In: Proceedings of the 10th International Conference on Software Engineering, Singapore, pp. 80–86 (1988)
36. Murphy, G.C., Notkin, D., Griswold, W.G., Lan, E.S.C.: An empirical study of static call graph extractors. ACM Trans. Softw. Eng. Methodol. **7**(2), 158–191 (1998)
37. MySQL: the world's most popular open source database. http://www.MySQL.org/
38. Neyman, J., Pearson, E.S.: On the use and interpretation of certain test criteria for purposes of statistical inference. Biometrika **20**, 175–240 (1928)
39. Nicolau, A., Li, G., Kejariwal, A.: Techniques for efficient placement of synchronization primitives. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, NC, USA, pp. 199–208 (2009)
40. Ocamlgraph: a graph library for Objective Caml. http://ocamlgraph.lri.fr/
41. OProfile—a system profiler for linux. http://oprofile.sourceforge.net/news/
42. Optimizing InnoDB disk i/o. http://dev.mysql.com/doc/refman/5.6/en/optimizing-innodb-diskio.html

43. org.apache.hadoop.io.compress.bzip2.CBZip2InputStream. http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/compress/bzip2/CBZip2InputStream.html
44. org.apache.hadoop.io.compress.bzip2.CBZip2OutputStream. http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/compress/bzip2/CBZip2OutputStream.html
45. Panda, P.R., Dutt, N.D., Nicolau, A.: Memory organization for improved data cache performance in embedded processors. In: Proceedings of the 9th International Symposium on System Synthesis, pp. 90–95 (1996)
46. Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J.: PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Toronto, ON, Canada, pp. 2–11 (2010)
47. Pearson, K.: On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arised from random sampling. Philosoph. Mag. Ser. **5**(50), 157–175 (1900)
48. Reiss, S.P.: The Field Programming Environment: A Friendly Integrated Environment for Learning and Development. Kluwer, Norwell, MA (1995)
49. Ryder, B.G.: Constructing the call graph of a program. IEEE Trans. Softw. Eng. **5**, 216–226 (1979)
50. Sereni, D.: Termination analysis and call graph construction for higher-order functional programs. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, Freiburg, Germany, pp. 71–84 (2007)
51. Shivers, O.G.: Control-flow analysis of higher-order languages of taming lambda. Ph.D. thesis, Carnegie Mellon University (1991)
52. Spearman, C.: The proof and measurement of association between two things. Am. J. Psychol. **15**, 72–101 (1904)
53. Spearman, C.: Footrule for measuring correlation. Br. J. Psychol. **2**(1), 89–108 (1906)
54. Spinellis, D.: Cscout: a refactoring browser for C. Sci. Comput. Program. **75**(4), 216–231 (2010)
55. Stube, A.O., Rexachs, D., Luque, E.: Software probes: towards a quick method for machine characterization and application performance prediction. In: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing, pp. 23–30 (2008)
56. SPEC CFP2006. http://www.spec.org/cpu2006/CFP2006/
57. SPEC CINT2006. http://www.spec.org/cpu2006/CINT2006/
58. SPEC OMP Benchmarks. http://www.spec.org/omp/
59. Tallent, N.R., Mellor-Crummey, J.M.: Effective performance measurement and analysis of multithreaded applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, NC, pp. 229–240 (2009)
60. The Caml Language. http://caml.inria.fr/
61. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Proceedings of the 15th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, Minneapolis, MN, pp. 281–293 (2000)
62. Valgrind. http://valgrind.org/
63. Wang, P.H., Collins, J.D., Wang, H., Kim, D., Greene, B., Chan, K.M., Yunus, A.B., Sych, T., Moore, S.F., Shen, J.P.: Helper threads via virtual multithreading on an experimental Itanium®2 processor-based platform. SIGPLAN Notices **39**(11), 144–155 (2004)
64. Zhang, W., Ryder, B.G.: Automatic construction of accurate application call graph with library call abstraction for java: research articles. J. Soft. Maint. Evol. Res. Pract. **19**(4), 231–252 (2007)