Journal of Systems Architecture 57 (2011) 121–133

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Efficient dynamic program monitoring on multi-core systems

Guojin He *, Antonia Zhai

Department of Computer Science and Engineering, University of Minnesota, MN, United States

ARTICLE INFO

Article history:
Received 26 March 2010
Received in revised form 15 July 2010
Accepted 26 July 2010
Available online 6 August 2010

Keywords:
Multi-core
Program monitoring
Compiler optimization

ABSTRACT

Dynamic program execution monitors allow programmers to observe and verify an application while it is running. Instrumentation-based dynamic program monitors often incur significant performance overhead due to instrumentation. Special hardware supports have been proposed to reduce this overhead. However, most of these supports often target specific monitoring requirements, and thus have limited applicability. Recently, with multi-core systems becoming mainstream, executing the monitored program and the monitor simultaneously on separate cores has emerged as an attractive option. However, due to the large amount of information forwarded to the monitor, existing approaches of dynamic monitoring on multi-core still suffers from significant performance overhead, unless adapt special-purpose hardware support.

In this paper, we present a novel dynamic execution monitoring model on the multi-core architecture. This model is based on two observations: (1) a monitor only requires specific information from the monitored program; (2) some information can be easily computed by the monitor from information that have already been communicated. Based on these observations, we propose hardware and software support to build a dynamic execution monitoring system that only communicates information that is relevant to monitoring purposes. Furthermore, we developed optimization techniques that decide the set of data to forward to the monitor and the data set to compute by the monitor, so that the total execution time of the monitor is minimized.

We evaluate the performance impact of the proposed dynamic program monitoring system with SPEC2006 integer benchmarks for two intensive monitoring tasks: taint-propagation and memory-bug-detection. Compared with instrumentation-based monitors, the proposed techniques is able to reduce the performance overhead of the two monitors by 3.7× and 2.2×, respectively.

1. Introduction

Monitoring software execution is crucial in software development for embedded systems. For example, test oracles monitor an application's execution by checking the outputs and part of the application's internal state; test adequacy coverage analysis tools detect what portion of an application has been executed and possibly how those parts of the application were reached; runtime security and safety monitors in mission critical systems must determine whether security and safety policies are maintained throughout the execution. These monitoring tasks generally require three crucial properties. First, to ensure that the performance of the monitored software is not degraded to a point where the monitoring is simply unfeasible, the monitoring activity must incur *low overhead*. Second, to monitor embedded applications, monitors must have *predictable behavior*, in terms of both functional behavior and performance, so that we can predict the impact of changed or removed monitors. Monitors are likely to change during the lifetime of the monitored software. For example, safety and security policies may change as the environment changes, especially in the case of test oracles and coverage measurement tools where monitor may be removed entirely at some point. Finally, to enable access to internal program state information, *fine-grained monitoring* is needed so that the state information in the monitored program as well as the program points where the monitoring takes place can be selected to suit the task at hand. To meet these requirements, traditional monitoring efforts typically require extensive instrumentation of the software and/or execution of the software in a dedicated hardware test-rig or emulator. Under such conditions, the software behavior is not the same as it would be in its intended target environment. To fully understand behaviors of software, in particular embedded software, the *production* software should be studied in its *production environment* with dynamic execution monitors.

One way to implement dynamic program monitoring is to use program instrumentation [1–3], where monitoring code is instrumented into the application program to closely observe the

Corresponding author. Tel.: +1 612 626 1285.
E-mail address: guojinhe@cs.umn.edu (G. He).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.07.011

execution progress. However, instrumentation-based monitoring often incurs high overhead, making this approach less attractive [2,3]. Moreover, instrumentation can create discrepancies between the monitored and the un-monitored programs. Correctness in the monitored program with instrumentation does not necessary imply the correctness of the un-monitored program. One example of instrumentation-based program monitoring is shown in Fig. 1(a). In this example, memory accesses through pointer are verified. Therefore, a call to the `VERIFY_LOAD()` function and a call to the `VERIFY_STORE()` function are inserted to check the addresses referred by pointer before actual memory operations happen.

To improve the performance of dynamic program monitoring, various hardware-based and software-based optimizations have been proposed [4–8,**?**]. However, these optimizations often aim at a specific monitoring task. For example, Raksha [8], LIFT [5] and FlexTaint [9] were only designed for taint-propagation tracking; and MemTracker [9] as well as HeapMon [7] were for memory-bug-detection. They are not generally applicable to meet diverse monitoring needs.

As the multi-core architecture becomes mainstream, a new approach for implementing dynamic program monitoring has been proposed [10,11]. This approach takes advantage of computing power provided by multi-core platforms and thus is able to support a large spectrum of monitoring tasks. In dynamic monitoring systems built on multi-core platforms, instead of instrumenting a program and executing it on a single core, a monitor is extracted from the original program and runs on a separate core. Two processes are generated in a monitored run for a program. The main process is the original program being monitored, which is referred to as the monitored process. The other process monitors the execution of the main process, and is referred to as the monitor. The monitored process continuously forwards computed results to the monitor via a communication queue to verify the execution progress. Based on this idea of decoupling monitored execution and monitoring work, Chen et al. [12] proposed a dispatch-based monitoring model in which the monitored code forwards all execution records to the monitor. The monitor runs a dispatch routine, which continuously reads the execution records forwarded from the monitored code, and determines what monitoring functions to invoke based on the record type. One example of such dispatch routine is shown in Fig. 1(b).

Although the performance overhead associated with monitors using the dispatch-based model is lower than that of instrumentation-based monitors, it is still fairly significant. The causes of the high performance overhead are the unnecessary communication and the dispatching cost related to unneeded messages. In a dispatch-based monitoring system, the monitored process may forward data that are unneeded by the monitor. In the example program in Fig. 1(a), if we are only interested in verifying memory operations by pointer, the monitor does not need execution results from other types of operations, thus communication for these results are unnecessary. Furthermore, dispatching a message consumes a certain number of CPU cycles in the monitor core. It involves decoding a message to get its type, extracting the value from the message, and jumping to the place where the appropriate monitoring is invoked. The type information is communicated solely for dispatching purpose and can be considered as unnecessary.

```
foo(int* p, int y) {
  int* q;
  int i,j,k;
  i = 0;
  do {
    q = p + i;
    VERIFY_LOAD(q);
    k = *q;
    j = k*k + i;
    VERIFY_STORE(q);
    *q = j + y;
    i++;
  } while(i<1024)
}
```

(a) Instrumentation-based monitor: calls to the verifying routines are inserted in the original program.

```
do {
  msg = receive();
  switch(msg.type) {
    ...
    case LOAD:
      VERIFY_LOAD(msg.body);
      break;
    case STORE:
      VERIFY_STORE(msg.body);
      break;
    default :
      break;
  }
} while(...)
```

(b) Dispatch-based monitor: dispatcher invokes the appropriate verification routine based on the forwarded record.

```
foo(int p, int y) {
  int q;
  i = 0;
  do {
    q = receive();
    VERIFY_LOAD(q);
    VERIFY_STORE(q);
    i = receive();
  } while(i<1024)
}
```

(c) Distill-based monitor (unoptimized): the monitor executes a distilled version of the program.

```
foo(int* p, int y) {
  int* q;
  p = receive();
  i = 0;
  do {
    q = p + i;
    VERIFY_LOAD(q);
    VERIFY_STORE(q);
    i++;
  } while(i<1024)
}
```

(d) Distill-based monitor (optimized): the monitor executes an *optimized* distilled version of the program.

Fig. 1. Four different implementations of the memory-bug-detection monitor. (a) Instrumentation-based monitor: calls to the verifying routines are inserted in the original program. (b) Dispatch-based monitor: dispatcher invokes the appropriate verification routine based on the forwarded record. (c) Distill-based monitor (unoptimized): the monitor executes a distilled version of the program. (d) Distill-based monitor (optimized): the monitor executes an *optimized* distilled version of the program.