Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 13-019

Efficient Test Coverage Measurement for MC/DC

Michael W. Whalen, Mats P. Heimdahl, and Ian J. De Silva

June 06, 2013

Efficient Test Coverage Measurement for MC/DC ⋆

Michael W. Whalen, Mats P.E. Heimdahl, and
Ian J. De Silva

Department of Computer Science, University of Minnesota

Abstract. Numerous activities require low-overhead monitoring of software applications, for example, run-time verification, test coverage measurement, and data collection. To support monitoring, current approaches usually involve extensive instrumentation of the software to be monitored, causing significant performance penalties and also requiring some means to ensure that the monitoring code will not cause incorrect behavior in the monitored application. To tackle this problem, we have explored a hardware-supported framework for monitoring and observation of software-intensive systems. In our approach, we leverage multi-core processor architectures to create a non-intrusive, predictable, fine-grained, and highly flexible general purpose monitoring framework. We have developed a novel architecture that augments each core with programmable extraction logic to observe an application executing on the core as its program state changes. Based on this architecture, we present a novel and highly efficient algorithm for tracking MC/DC coverage.

1 Introduction

Numerous and diverse software engineering activities necessitate monitoring of an application of interest. For example, run-time security and safety monitors in critical systems can determine whether security and safety policies are maintained by an application [13], and test adequacy coverage analysis tools must monitor an application to determine which portions of an application have been executed (and possibly how those parts of the application were reached). To be practically useful, these tasks typically require low overhead monitoring to ensure that the performance of the monitored software is acceptable.

Different monitoring approaches have been pursued in different communities to address performance, genericity, and cost concerns. Test monitoring oracles are generally ad hoc and rely on intrusive software instrumentation [14]; run-time verification typically relies on software instrumentation with significant performance overhead for the target application [13], and most of the dedicated hardware solutions are targeted towards the monitoring for narrow properties [4, 22, 2, 15, 23, 3, 19].

⋆ This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and NSF grants CCF-0916583 and CNS-0931931.

To alleviate the problems with performance and narrow specialization of instrumentation for monitoring purposes, we have pursued an approach leveraging the rapid emergence of multi-core processor architectures [5, 10, 1, 20, 11]. This approach achieves a non-intrusive, predictable, fine-grained, and highly flexible general purpose monitoring framework through *monitoring-aware* compilers coupled with *novel architectural enhancements* to the multi-core architectures. We use separate cores for the execution of the application to be monitored and the monitors. Previously, He et al. have augmented each core with identical programmable *extraction logic* that can observe an application executing on the core as its program state changes [9, 7]. If a state change that needs to be monitored occurs, the extraction logic will pack the state change into a message and send it to one of the *monitor cores* for verification.

In this architecture, one or more monitoring cores can be used to share the monitoring workload and significantly reduce monitoring overhead. For many monitoring tasks, no instrumentation is required of the monitored application, so it is possible to add monitors without any impact (other than possibly run time) to existing applications, easing certification concerns with respect to monitoring of critical applications.

This multi-core architecture has applications in many software engineering tasks. In this report we evaluate the architecture on one challenging monitoring task—determining Modified Decision and Condition Coverage (MC/DC) over C code. Even with dedicated hardware support, this monitoring task poses challenges at two levels: (1) the monitoring takes place at the object code level whereas the MC/DC criterion is defined over the source code—a non-trivial mapping must be made and (2) the algorithm keeping track of the MC/DC coverage is surprisingly complex and must be highly optimized as to not incur undue overhead in the monitoring process. We have addressed both challenges in our work and conducted an experiment to evaluate the efficacy of our approach. Our work shows that this monitoring can be achieved with little overhead (from 0% to 43%) on example systems.

The remainder of the paper is organized as follows. We provide the background for our work an overview of the architecturally supported vision in Section 2. We present the details of the efficient MC/DC marking algorithm in Section 3. Finally, in Section 4, we discuss future directions and conclude our discussion of the implications of our results.

2 Background

We outline two relevant areas of research. First, we provide a short overview of hardware supported monitoring and our purposed architecture. Second, we discuss the monitoring for test adequacy coverage—in particular, the MCDC criterion.

2.1 Hardware Supported Multi-Core Monitoring

A monitor observes the internal states of an application and performs some computation on the information, for example, to verify a property at run-time or measure test coverage during testing. Monitors can be invoked in two distinct ways: *explicit* and *implicit* invocation. Monitors with *explicit invocation* are invoked when the execution of the application reaches a certain program point; examples are monitors performing pre- and post-condition checking that must be invoked at the entry and exit of a function. Monitors with *implicit invocation* are invoked when an application reaches a state of interest to the monitor; an example would be a monitor checking a program invariant.

The simplest way to monitor the execution of an application is to integrate the set of monitors with the application through instrumentation [13, 12]. This way, the application state is completely visible to the monitor; no explicit state extraction and communication is needed. The monitor maintains its own state in the same address space as the application. In this scenario, dispatching *explicit* monitors is trivial—calls to the appropriate monitor can be insert as instrumentation in the original program. To dispatch the appropriate *implicit* monitors, all instructions that can generate a state change of interests must be instrumented. At runtime, all relevant state changes must be examined, and the proper monitor dispatched. If instrumented instructions occur often, this can be a source of significant performance overhead. One way to mitigate the performance degradation is to migrate the monitor to a separate processor [18, 8]. This will allow the monitor and the application to execute in parallel and—if the monitor can keep up with the application—the performance penalties are now limited to the overhead associated with extraction and communication of state information and competition for shared resources.

Recently, there have been proposals for hardware support for a variety of monitoring activities; in particular to support fine-grained monitoring. Nevertheless, most of these proposals support one narrow type of monitor, such as monitoring memory bugs [17, 22] or taint analysis [4, 22, 19, 3, 16, 21]. These solutions can provide extremely low overhead monitoring, but they are all targeted to specific monitoring tasks and allow little or no customization.

There are four steps in monitoring: (1) extract the desired information from the core executing the application, (2) forward it to the monitor core, (3) update the monitor core state with the new information, and (4) dispatch the appropriate monitors. All four steps must be supported with hardware to provide the performance needed. In particular, hardware support for information extraction is essential since our aim is to avoid instrumenting the application program. To this effect, He et al. have in previous work integrate an *extraction logic* onto each core, and have the monitor configure the extraction logic with an event-of-interest list when the program is loaded [9, 7]. This extraction logic is capable of capturing a variety of instruction-level events. For example, the extraction logic can be configured to detect execution of specific instruction types—such as a conditional jump—or access to a memory or register allocated variable.

A	B	A *and* B
T	T	T
T	F	F
F	T	F

Table 1. Example of a test suite that provides MCDC over A *and* B

A	B	C	A *or* $(B$ *and* $C)$
F	T	T	T
F	T	F	F
F	F	T	F
T	F	T	T

Table 2. Example of a test suite that provides MCDC over A *or* $(B$ *and* $C)$

Once such events are detected, information about the event are forwarded to the monitor.

2.2 Modified Condition/ Decision Coverage Criteria

Modified Condition/ Decision Coverage (MCDC) [6] is an important structural coverage criteria used in the avionics software domain. This metric is designed to demonstrate the independent effect of basic Boolean conditions (i.e., subexpressions with no logical operators) on the Boolean decision (expression) in which they occur. A test suite is said to satisfy MCDC if executing the test cases in the test suite will guarantee that:

- every point of entry and exit in the model has been invoked at least once,
- every basic condition in a decision in the model has taken on all possible outcomes at least once, and
- each basic condition has been shown to independently affect the decision's outcome

where a basic condition is an atomic Boolean valued expression that cannot be broken into Boolean sub-expressions. In this paper we use *masking* MCDC [6] to determine the independence of conditions. In masking MCDC, a basic condition is *masked* if varying its value cannot affect the outcome of a decision due to structure of the decision and the value of other conditions. To satisfy masking MCDC for a basic condition, we must have test states in which the condition is not masked and takes on both True and False values. Thus, a pair of test cases must exist for each basic condition in the test-suite to satisfy MCDC. Note that each instance of a condition within a formula is treated separately: the formula `A and (B or A)` has three basic conditions, and we must determine the independence of each instance of A separately.

To better illustrate the definition of masking MCDC, consider the expression `A and B`. To show the independence of `B`, we must hold the value of `A` to True; otherwise varying `B` will not affect the outcome of the expression. Independence of `A` is shown in a similar manner. Table 1 shows the test suite required to satisfy MCDC for the expression `A and B`. When we consider decisions with multiple Boolean operators, we must ensure that the test results for one operator are not masked out by the behavior of other operators. For example, given `A or (B and C)` the tests for `B and C` will not affect the outcome of the decision if `A` is True. Table 2 gives one of the test suites that would satisfy MCDC for the expression `A or (B and C)`.

A	B	A *and* B
T	T	T
T	F	F
F	$*$	F

A	B	A *or* B
T	$*$	T
F	T	T
F	F	F

Table 3. Short circuit MCDC obligations for *and*, *or*

In C/C++, Java, and most other imperative languages, Boolean expressions are evaluated using short circuit evaluation. In short circuit evaluation, the right-side subexpressions of `and` or `or` expressions are not evaluated if the expression is known to be TRUE or FALSE after evaluation of the left-side subexpression. This occurs when the left side of an `or` expression evaluates to TRUE or the left side of an `and` expression evaluates to FALSE, respectively. In these instances, it is not possible to determine whether the right side subexpression could mask out the value of the left-side expression—the right side subexpression is not evaluated—so the right-hand side value becomes a 'don't care' (denoted '*'), as shown in Table 3. Since we are analyzing imperative code, we will assume the short-circuit version of MCDC in subsequent sections.

3 Marking Function Generation

Recall that the independent effect of each condition is demonstrated when the condition is not masked while taking on both TRUE and FALSE values. Therefore, we use an array called an *independence array* containing two bits for each basic condition within the program. The bits are set to true when the condition has been shown to independently affect the outcome of its decision while evaluating to TRUE and FALSE, respectively. If all bits are set, we state that complete MCDC coverage has been achieved. We define partial coverage as the elements set to true divided by the number of all elements in the independence array.

C/C++ use short circuit evaluation, and evaluation of decisions must occur in the left-to-right order of the constituent conditions (though some conditions may not be evaluated). Determining the MCDC independence of a condition therefore depends on the location of the condition within the decision. Conditions on the right side of Boolean operators may mask out the effect of left-side conditions but not vice-versa. On the other hand, conditions on the left side of Boolean operators may cause the evaluation of right side conditions to be skipped entirely (short-circuited). For example, consider Figure 3, which shows a complex Boolean decision and its translation into SPARC assembly code. If `(a == 5)` is false, then we will skip the evaluation of `(b || c)` entirely. If we evaluate `c`, then it must be the case that `b` evaluated to FALSE and that `(a == 5)` evaluated to TRUE. If `c` evaluates to FALSE, then `(b || c)` must evaluate to false, and the effect of `(a == 5)` is masked out.

As we evaluate conditions left-to-right within a decision, we want to consider their independent effect on the decision. By evaluating the condition, we know that it is *relevant* to the decision, but we do not necessarily know that it has an

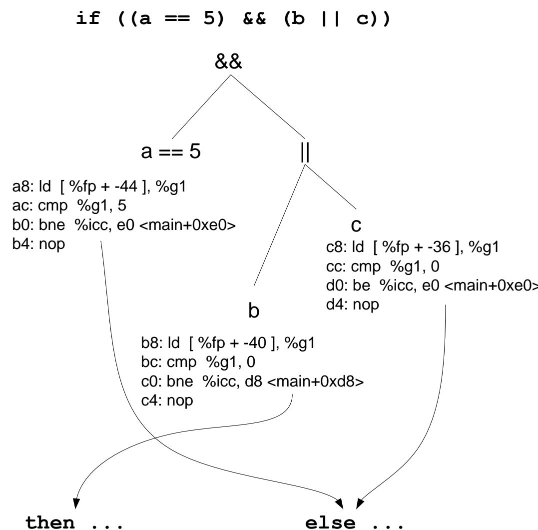

Fig. 1. Code snippet with complex decision and its translation into assembler

independent effect, because its effect may be masked out by a condition evaluated subsequently within the same decision. By the same token, the evaluation of the current condition may mask out the effect of earlier conditions within the decision. To this end, for each condition within a decision, we define a *marking function* that exploits the short-circuit structure of the decision. This marking function does three things:

1. it *sets* the true/false independence bit associated with the condition in a temporary bit array depending on the outcome of the condition.
2. If the outcome of the condition leads to masking, it may *clear* bits associated with previous conditions within the temporary bit array
3. if the outcome of the condition causes the outcome of the decision to be known, it *latches* the result of the temporary array into the independence array.

The marking procedures operate over a temporary array until the outcome of the decision is known, at which time we latch it into the independence array. As we will see later, these tasks can be accomplished by a handful of bit masking operations, so the marking procedures can be very efficient.

The marking procedure can be used to create an efficient single-core algorithm by instrumenting the source code to call the function after evaluation of each condition. Nevertheless, a still more efficient implementation can be created by moving the monitoring code to a separate core. Here, we must map the source code conditions to instructions in the generated assembly code that can be observed using our multicore framework.

For the SPARC architecture, monitoring the value of conditional branch instructions is sufficient for the alternate core. An example of a complex decision

and its translation into SPARC assembly by the GCC 4.1.2 compiler is shown in Figure 3. Notice that the structure of the original decision is translated into a block of assembly code with conditional branch instructions for each condition. The conditional branch instructions are ordered sequentially matching the left-to-right order of the conditions. If the value of a condition causes a subexpression to become known, the conditional branches "jump out" of that subexpression to the next unknown part. For example, if the expression `(a == 5)` is FALSE then the value of the `if` decision is known to be FALSE and the assembly code immediately jumps to the `else` branch. Similarly, if the value of `(b)` is TRUE, the code immediately jumps to the `then` branch.

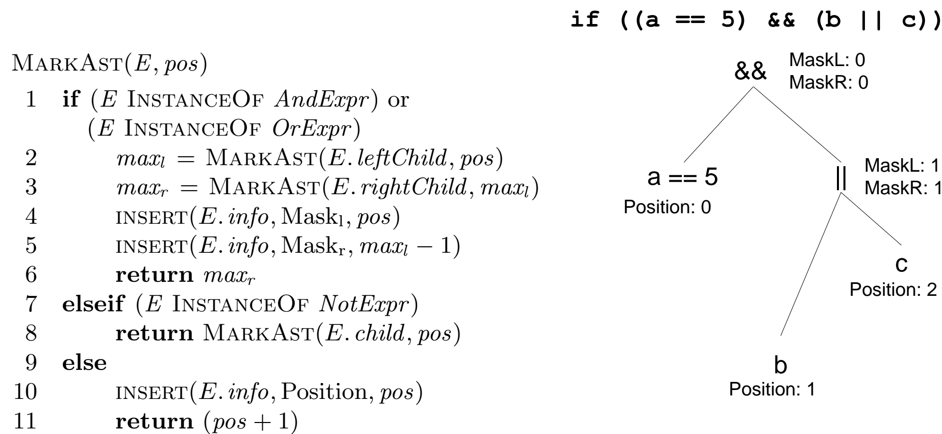

Fig. 2. (a) AST annotation function (b) Annotated code example

Pseudocode for the marking function generation is shown in Figures 2, 3, and 4. The MARKAST function (Figure 2) annotates the abstract syntax tree (AST) with a position number for each condition within the decision and two masking range values for each binary Boolean operator. The masking bits simply define the range of position numbers assigned by the left side of the operator - this can be used to determine the set of bits that can be masked if the right side of the operator has the "wrong" value. The arguments to MARKAST are E, an AST expression and *pos*, the initial bit position to use for allocating condition bits. The function returns the next unused position after all conditions have been allocated position bits. An example of a marked tree is shown in Figure 2(b).

The second pseudocode figure (Figure 3) defines functions that determine whether a condition masks out prior conditions within the decision. In short circuit MCDC, as described in Table 3, masking only occurs when the right side of an `and` expression is FALSE or the right hand side of an `or` expression is TRUE. In these cases, we want to clear any of the bits from the left-hand side.

```
opVal(N, v)
1  if (N.side == Right) return v
2  elseif ((N.expr InstanceOf AndExpr) and (v == False)) return False
3  elseif ((N.expr InstanceOf OrExpr) and (v == True)) return True
4  elseif ((N.expr InstanceOf NotExpr) and (v == True)) return False
5  elseif ((N.expr InstanceOf NotExpr) and (v == False)) return True
6  else return Unknown
```

```
setMask(path, mask, v)
1  if ((isNil(path)) or (v == Unknown)) return mask
2  else
3        N = Head(path)
4        info = N.expr.info
5        tl = Tail(path)
6        if (N.side == Right) and
           (((N.expr InstanceOf AndExpr) and (v == False)) or
           ((N.expr InstanceOf OrExpr) and (v == True)))
7              mask = addToMask(mask, lookup(info, Mask_l), lookup(info, Mask_r))
8        return setMask(tl, mask, opVal(N, v))
```

Fig. 3. Pseudocode for masking functions

Note that in a complex decision, a condition may be both on the left side of one operator and the right side of another (for example, the condition `b` in Figure 3). In this case, even if the condition is on the left-hand side of its immediate parent, it may mask out conditions further up within the decision tree structure. We recursively call the setMask function until the value of the current operator is Unknown. In order to make this determination, we must know the ancestor expressions of the condition being evaluated. These ancestors are stored in a *path* that defines the sequence of expressions from the root expression of the decision down to the current condition and, for binary operators, the side (left or right) of the next element on the path. The type for paths is defined as follows:

```
1  type PathElemType == record of
2        expr: Expr
3        side: enum {Left, Right, Unary}
4  end record
5
6  type PathType == list ofPathElemType
```

The setMask function takes as arguments (1) *path*, the path from the root of the decision to the current subexpression, (2) *mask*, the current set of bits that can be masked by this condition, and (3) v, the assumed value of the expression. Given the "known" value of a subexpression, we can traverse *path* upwards to determine which bits to mask. As we traverse upwards, we will either reach a

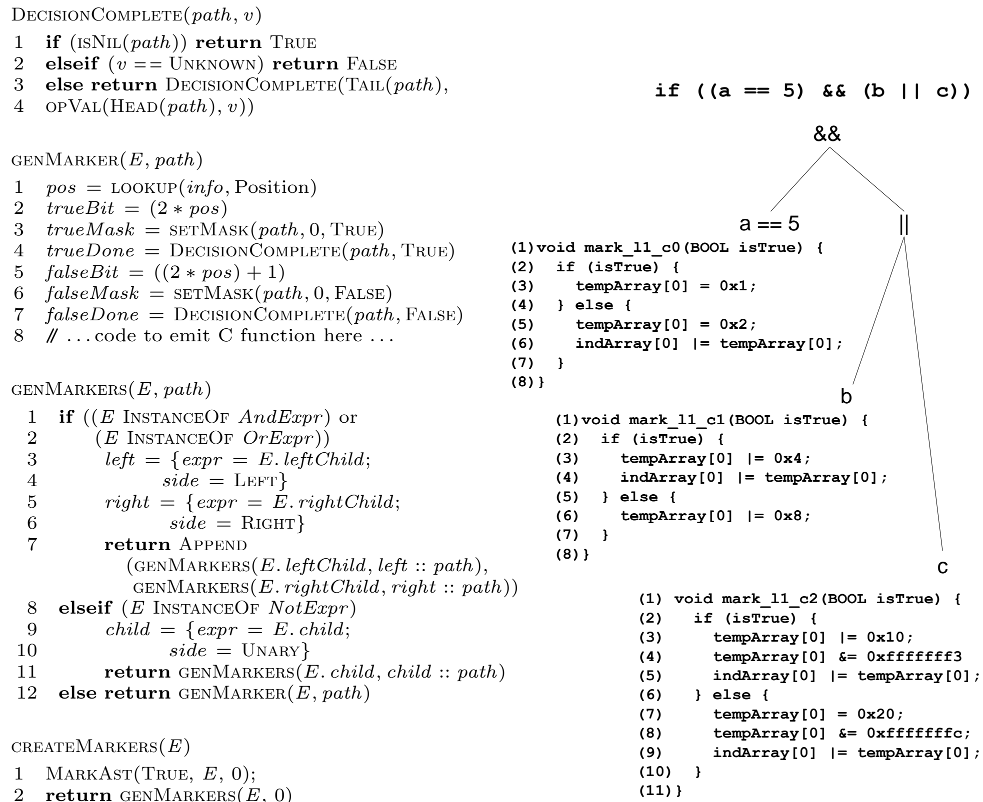

Fig. 4. (a) Pseudocode to generate marking functions, (b) Examples of generated marker functions

point where the value of an ancestor expression cannot be determined or we reach the top of the decision. In either of these cases, there are no further bits that can be masked out, so we terminate.

The third figure (Figure 4) provides the functions to generate the masking procedures. Starting from the top-level function createMarkers, we first annotate the position information in the decision (calling MarkAst), then call genMarkers to actually create the condition functions. genMarkers traverses the expression structure and calls genMarker to create a marking function for each leaf-level condition. It then appends all of these functions into a list. It takes as arguments E, the current expression being examined and the *path* from the root of the decision. genMarker creates the C function for each condition, using the functions defined previously to determine the bit to set and also whether any bits are to be masked out based on the assignment to the condition. Examples of the generated marker functions are shown in Figure 4 (b). For each truth value, the marking function marks a bit into the temporary array, may mask out earlier conditions and, if the value of the decision is determined after evaluation, latch the temporary array into the independence array. For example, lines (3-5) of function *mark_l1_c2* perform each of these tasks.

4 Conclusion

In this paper, we have proposed an efficient approach for real-time monitoring of the MC/DC test metric. Our approach is based on monitoring of conditional branch instructions in the running application. It requires only a handful of instructions per condition to monitor, Although our work on a monitoring aware compilers and multi-core architecture is far from complete, the initial steps and performance evaluation presented in this report illustrate the potential for this approach as we attempt to make run-time verification and monitoring in the *production* environment standard practice. Based on this experiment, we believe that further optimizations of the monitoring code can further lower monitoring overhead.

References

1. AMD Corporation: Leading the industry: Multi-core technology & dual-core processors from amd. http://multicore.amd.com/en/Technology/ (2005)
2. Chen, S., Falsafi, B., Gibbons, P.B., Kozuch, M., Mowry, T.C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G.R., Lin, B., Schlosser, S.W.: Log-based architectures for general-purpose monitoring of deployed code. In: ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability. pp. 63–65. ACM, New York, NY, USA (2006)
3. Crandall, J.R., Chong, F.T.: Minos: Control data attack prevention orthogonal to memory model. In: MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture. pp. 221–232. IEEE Computer Society, Washington, DC, USA (2004)

4. Dalton, M., Kannan, H., Kozyrakis, C.: Raksha: A flexible information flow architecture for software security. In: 34th Annual International Symposium on Computer Architecture (ISCA '07) (2007)
5. Friedrich, J., McCredie, B., James, N., Huott, B., Curran, B., Fluhr, E., Mittal, G., Chan, E., Chan, Y., Plass, D., Chu, S., Le, H., Clark, L., Ripley, J., Taylor, S., Dilullo, J., Lanzerotti, M.: Design of the POWER6(TM) Microprocessor. In: 2007 IEEE International Solid-State Circuits Conference (Feb 2007)
6. Hayhurst, K., Veerhusen, D., Rierson, L.: A practical tutorial on modified condition/decision coverage. Tech. Rep. TM-2001-210876, NASA (2001)
7. He, G., Zhai, A.: Improving the performance of program monitors with compiler support in multi-core systems. In: The IEEE International Parallel & Distributed Processing Symposium (IPDPS) (2010)
8. He, G., Zhai, A., Yew, P.C.: Ex-mon: An architectural framework for dynamic program monitoring on multicore processors. In: The Twelfth Workshop on Interaction between Compilers and Computer Architectures (Interact-12) (2008)
9. He, G., Zhai, A., Yew, P.C.: Ex-mons: An architectural framework for dynamic program monitoring on multicore processors. In: The 12th Workshop on Interaction between Compilers and Computer Architectures (INTERACT 12) (2008)
10. Intel Corporation: Intel's dual-core processor for desktop PCs. http://www.intel.com/personal/desktopcomputer/dual_core/index.htm (2005)
11. Intel Corporation: Intel itanium architecture software developer's manual, revision 2.2. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm (2006)
12. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: The 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200 (June 2005)
13. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN 07 Conference on Programming Language Design and Implementation (PLDI'07). San Diego, California, USA (Jun 2007)
14. Pezze, M., Young, M.: Software Test and Analysis: Process, Principles, and Techniques. John Wiley and Sons (October 2006)
15. Qin, F., Lu, S., Zhou, Y.: Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In: 11th International Symposium on High-Performance Computer Architecture (HPCA-11) (Feb 2005)
16. Qin, F., Wang, C., Li, Z., seop Kim, H., Zhou, Y., Wu, Y.: Lift: A low-overhead practical information flow tracking system for detecting security attacks. Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on pp. 135–148 (Dec 2006)
17. Shetty, R., Kharbutli, M., Solihin, Y., Prvulovic, M.: Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. IBM J. Res. Dev. 50(2/3), 261–275 (2006)
18. Shimin Chen, Michael Kozuch, T.S., Falsafi, B., Gibbons, P.B., Mowry, T.C., Ramachandran, V., Ruwase, O., Ryan, M., Vlachos, E.: Flexible hardware acceleration for instruction-grain program monitoring (June 2008)
19. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems. pp. 85–96. ACM, New York, NY, USA (2004)
20. Sun Corporation: Throughput computing—niagara. http://www.sun.com/processors/throughput/ (2005)

21. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: Flexitaint: A programmable accelerator for dynamic taint propagation. In: 14th International Symposium on High-Performance Computer Architecture (HPCA-14) (Feb 2008)
22. Venkataramani, G., Roemer, B., Solihin, Y., Prvulovic, M.: Memtracker: Efficient and programmable support for memory access monitoring and debugging. In: 13th International Symposium on High-Performance Computer Architecture (HPCA-13) (Feb 2007)
23. Zhou, P., Qin, F., Liu, W., Zhou, Y., Torrellas, J.: iwatcher: Simple, general architectural support for software debugging. In: 31st Annual International Symposium on Computer Architecture (ISCA '04) (2004)