NUDA: A Non-Uniform Debugging Architecture and Nonintrusive Race Detection for Many-Core Systems

Chi-Neng Wen, Shu-Hsuan Chou, Chien-Chih Chen, and Tien-Fu Chen

Abstract—Traditional debugging methodologies are limited in their ability to provide debugging support for many-core parallel programming. Synchronization problems or bugs due to race conditions are particularly difficult to detect with existing debugging tools. Most traditional debugging approaches rely on globally synchronized signals, but these pose their own problems in terms of scalability. The first contribution of this paper is to propose a novel non-uniform debugging architecture (NUDA) based on a ring interconnection schema. Our approach makes hardware-assisted debugging both feasible and scalable for many-core processing scenarios. The key idea is to distribute the debugging support structures across a set of hierarchical clusters while avoiding address overlap. The design strategy allows the address space to be monitored using non-uniform protocols. Our second contribution is to propose a nonintrusive approach to lockset-based race detection supported by the NUDA. A non-uniform page-based monitoring cache in each NUDA node is used to keep track of the access footprints. The union of all the caches can serve as a race detection probe without disturbing execution ordering. Using the proposed approach, we show that parallel race bugs can be precisely captured, and that most false-positive alerts can be efficiently eliminated at an average slowdown cost of only 1.4-3.6 percent. The net hardware cost is relatively low, so that the NUDA can easily be scaled to increasingly complex many-core systems.

Index Terms—NUDA, lockset, data race, nonintrusive, manycore, debugging.

1 INTRODUCTION

MULTITASKING concurrency in the context of parallel programs is essential to the future development of MPSOC. However, nondeterministic parallel programs are hard to debug with traditional methodologies because subsequent executions with identical inputs are not guaranteed to result in the same behavior. Using a software instrument or debugger to insert probe code into the target program can help to identify potential problems, but it may cause a probe effect.

Many-core debugging poses several challenges.

First, the debugging or monitoring cannot disturb the sequential consistency of the target program. This kind of disturbance is called a "probe effect," [1] and will tend to mask certain synchronization errors after an unknown delay.

Second, in the context of parallel sequencing, on-chip trace modules are usually employed to record program execution or memory/register state changes and to allow for reliable offline reexecution using this information. A tracer requires a large on-chip trace buffer, as well as a global timestamp to correlate traces across different cores.

Third, bugs due to unsynchronized access to shared memory in parallel programs (or so-called data race) are even harder to detect using traditional debuggers; even the on-chip trace modules are supported. A lengthy trace may be required in order to observe a race condition that results from multiple concurrent accesses. Instead of fully tracing the entire execution, we advocate detecting data races at runtime, and alerting the debugger/monitor to record traces for debugging replay.

Fourth, most debugging functions are based on watching instructions and data addresses generated by the processors. The debug engine must constantly check with either breakpoint or watchpoint registers. A many-core system makes the debugging even more complicated.

Finally, traditional debugging facilities rely on globally synchronized signals and are not scalable to many-core systems. Synchronization problems or bugs due to race conditions are particularly difficult to detect. In addition to address comparisons using breakpoints and watchpoints, data race detection involves the extra requirement to watch many more lock/unlock and shared addresses at the same time.

Hence, this paper proposes a "Non-Uniform Debugging Architecture" (NUDA) to solve the above challenges in the context of many-core debugging.

The key idea of the NUDA is that we distribute the monitoring data and instruction addresses in a hierarchical manner using a non-uniform address space, caching frequent addresses (page-based) with distributed content addressable memory (CAM) tables, and thereby avoiding the need for centralized address comparisons. Once an alert event is detected, the NUDA employs a synchronization-token mechanism to notify each of the processor cores to stop or to restart separately without relying on a synchronized global signal. With the NUDA's distributed notification

- *C-N. Wen and S-H. Chou are with the Department of Computer Science and Information Engineering, National Chung-Cheng University, 168, University Rd., Min-Hsiung Township, Chia-Yi 722, Taiwan. E-mail: {dave.tw, csh93chou}@gmail.com.*
- *C-C. Chen and T-F. Chen are with the Department of Computer Science, National Chiao-Tung University, 1001 University Road, Hsinchu 300, Taiwan. E-mail: {john740207, tfuchen}@gmail.com.*

Manuscript received 25 Jan. 2010; revised 29 July 2010; accepted 17 Nov. 2010; published online 8 Dec. 2010
Recommended for acceptance by D. Gizopoulos.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2010-01-0055.
Digital Object Identifier no. 10.1109/TC.2010.254.

0018-9340/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

abilities, we can implement various debugging facilities to watch, detect, or even capture the execution trace in order to help develop parallel programs in many-core systems.

This work also proposes a programmable debugging methodology, by which users specify anchors and execution sequences to ensure that interactions between parallel programs are executed in the correct sequence. The performance slowdown associated with the NUDA is almost negligible, because intra-cluster debugging event checks are more frequent than inter-cluster interactions. We demonstrate that the target system can remain nonintrusive when being debugged and probed for synchronization problems using the proposed NUDA mechanism.

In summary, the contributions of this work are as following:

1. An isolated nonintrusive debugging architecture based on a non-uniform memory mapping.
2. A distributed and simultaneous notify mechanism (Sync-token) upon NUDA for synchronous debugging-control.
3. Software programming model (RunAssert) for many-core debugging with NUDA.
4. A lockset-based data rece detection design on NUDA.

2 Related Work

2.1 Debugging and Architecture

Recently, the number of cores is increasing rapidly. For instance, IBM Cell is a heterogeneous multicore system, the Intel Larrabee [2] is an x86-based many-core system, and the nVidia Fermi architecture [3] has $1{,}000+$ cores in a single chip. No matter how complicated those systems are and how simple the applications on them, the software developer must inevitably face that long-term stage of the development cycle known as the debugging stage.

There are many hardware approaches [4], [5] to help with debugging. Based on the previous works, this work focuses on a nonprobe effect environment for automatic parallel program fault detection. The standard data path [6], [7] performs adequately and is popular and well known. However, the protocol and configuration of those debug buses do not have enough flexibility and capability to handle manycore's debugging information.

To address this issue, OCP-IP [6] announced an independent debugging bus and corresponding core socket for industry many-core debugging. The OCP-IP's core socket provides the multicore system with an isolated cross trigger and tracing channel. Therefore, OCP-IP provides a standard core interface for debugging purposes, regardless of the debugging channel is or whether the core type is heterogeneous. Contemporary embedded processor paradigms such as ARM CoreSight [8] and MIPS On-Chip Instrumentation [9] feature nonintrusive trace modules embedded within the processors.

Architectural supports for runtime debugging are increasingly important for future many-core systems, especially for detecting race conditions or deadlock. Those hardware approaches are usually monitor-based models [10], [11] which are consuming lots of memory. Therefore, the concept of non-uniform cache architecture (NUCA) is originally used for L2 cache sharing in many-core systems [12], because it represents a distributed sharing mechanism with non-uniform memory access latency. The NUCA takes advantage of high data locality to shorten access latency and low cache-miss rates, without cache-coherency overhead. In this work, we take similar design concept for supporting runtime debugging.

2.2 Race Detection and Related HW Approaches

There are two algorithms to detect race conditions, the "happened-before" [13], [14] and "lockset-based" [15], [16], [17] algorithms. The happened-before algorithm works by comparing the timestamps or vector clocks [18] of particular synchronization operations, such as access to shared objects, between threads. The benefit of the happened-before algorithm is the fact that it is easily for hardware implementation. However, the drawbacks are that it only discovers those races that are manifested during the monitored execution. On the other hand, the lockset-based algorithm provides more precise information to overcome the limitations of the happened-before algorithm. The Eraser [15] probe all of the lock/unlock operations and memory access events, in order to detect data races efficiently.

There are several hardware approaches for debugging. FDR [19] and Rerun [20] are hardware approaches that use a low-overhead hardware recorder in the context of caches or cores, essentially to log the minimum thread ordering information that is necessary to play back the multiprocessor execution faithfully after the event. It truly helps programmers to debug, but a lengthy trace may be required. HARD [11] provides a novel hardware-assisted lockset-based race detection approach, and it employs additional fields (such as a Bloom filter vector(BFVector)) in the cache and detects the race condition. The HARD [11] is the first hardware feasible by efficiently storing and maintaining the candidate set, which is a set of locks protecting a variable in hardware, and radically simplifying the set operations in the lockset algorithm. However, because the design is architecture-dependent (passive), it is subject to the following imperfections of degraded race detection capability and high memory use:

1. Serious false negatives caused by cache miss.
2. False negatives caused by collisions of the Bloom filter.
3. Architecture dependence and low scalability.
4. Large memory cost in L1 and L2 cache hierarchies.
5. Incomplete barrier handling.

Those imperfections debase the reliability and scalability of the hardware race detection.

3 Nonuniform Debugging Architecture

Most of today's many-core debugging is trace-based and relies on the offline verification of history or execution sequencing. The drawback of this methodology is an unacceptably large storage requirement and slow operating cycles. In order to support versatile real-time debugging requirements, the NUDA is a technique that has been proposed for debugging sequentially consistent parallel

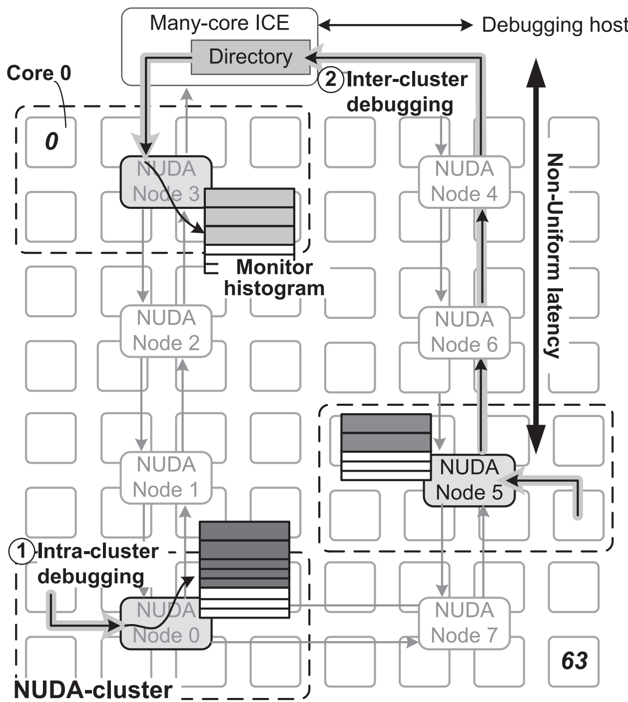


Fig. 1. The NUDA system architecture.

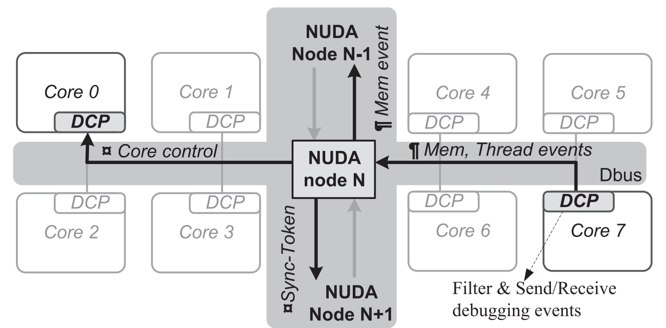


Fig. 2. The concept of the debugging cluster (NUDA cluster).

programs on many-core systems. It has a flexible and configurable infrastructure to handle growing numbers of cores in the future. However, debugging events are usually triggered by core/memory/device components, and too many unnecessary events will cause a traffic jam of the NUDA's interconnections and degrade the system performance. Moreover, unnecessary events may also intrude in the histogram-table in the NUDA node and lower the debugging capacity. Therefore, an efficient and hardware-feasible event filter is urgently required. Race detection, nearly unlimited break/watchpoints and cross-event trigger are critical for multicore/multithreading real-time debugging, and are all relative to memory access to a certain address.

Fig. 1 shows the NUDA system architecture, with a 64-core system as a fundamental platform. Without loss of granularity, the fundamental platform is unrestricted in its memory architecture, interconnection between cores, I/O facilities, etc. In fact, the NUDA is a scalable nonintrusive subsidiary architecture that does not share any resources with the fundamental platform. In brief, the NUDA framework contains three major parts: the NUDA architecture itself, the sync-token protocol on the NUDA and the related software solutions.

The NUDA architecture consists of a cluster with a NUDA node, the NUDA interconnection to transfer information between NUDA nodes, and the non-uniform memory shared with NUDA nodes. Due to the distributed coordinate architecture, the philosophy of the NUDA defines the cluster as a collaborative unit.

Fig. 2 shows the concept of a NUDA cluster that features two to eight neighboring cores and allows communication with the related NUDA nodes (Node $N-1$ and Node $N+1$) by an independent debugging channel (the vertical gray bar) and shared non-uniform memory. System designers can contribute their target system by clusters. The use of clusters offers the following benefits. First, a structure design is more flexible to face different system configurations and increasing numbers of cores. Second, we can classify the debugging into two categories, the intracluster and intercluster debugging events. The intracluster debugging events represent those of temporary/special locality and we can handle them immediately; in terms of the concept of time-to-space, the intercluster debugging events happen infrequently. Finally, from the perspective of the trend of parallel software structure, let the related tasks aggregate into a clusters provide better performance [21] and programmability [22].

A NUDA node is capable of gathering and organizing related information as monitored records. It also provides a programmable way for the user to reconfigure it, such as user-defined assertions. Moreover, a NUDA node handles the search/migrate/update to the non-uniform memory. From the programmer's viewpoint, a NUDA node is like a simple subsystem that contains a NUDA node controller with a separated local memory and a global memory (non-uniform memory) system. As shown in Fig. 2, the debugging coprocessor (DCP) plays the role of the interface between the fundamental platform and the NUDA. Massive histogram information is gathered by the DCP on each core and then sent to the NUDA node. Hence, the DCP should be lightweight and filterable. In contrast to the NUDA node, the DCP receives the NUDA node's command packet, decodes it, and then invokes the indicated debugging operations, such as stop/step/continue executing a core.

A NUDA also contains flexible interconnections that can be extended as the system grows. In this work, we use a ring as the interconnection structure. The reason for using the ring interconnections is not only to reduce costs but also to take advantage of the ring broadcast mechanism for the synchronization of core debugging events.

In addition, most of the essential debugging functions are based on histogram data comparison. A larger storage space is acquired as the system grows. As shown in Fig. 1, the NUDA is based on distributed cluster architecture; this work uses a local storage in each NUDA node and then unifies those local storages into non-uniform memory architecture (NUMA), which is totally isolated from the memory system of the fundamental platform. First, the local storage is well prepared by a CAM as a map for fast intracluster event checking. Second, each NUDA node's local storage is a page-based non-uniform memory, and each page in this non-uniform memory is mapped into a global dictionary in the

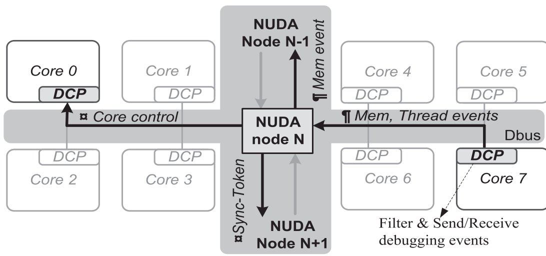


Fig. 3. The structure of the NUDA node.

many-core ICE in case of an intracluster check failure. Once the dictionary indicates where the page is, the intercluster event checking will be invoked automatically.

There are three main motivations to use non-uniform memory as the NUDA's storage system:

1. Non-uniform memory is highly extendable, as more clusters in system, and hence more memory space, becomes available.
2. Non-uniform memory has at least one data copy in this work; we don't have to save a redundant histogram in several pages. The proportionality of memory cost and memory capacity for monitoring is the most suitable for debugging purpose.
3. Non-uniform memory access operation is efficient. Accessing the local pages is faster than accessing the remote pages in the other NUDA nodes. Moreover, the data migration can dynamically revise the page locations and thus promote access efficiency.

3.1 The Structure of the NUDA Node

As shown in Fig. 3, the NUDA node structure consists of three major parts: computing, memory, and communication. The specific router handles a NUDA node's communication with its neighboring NUDA nodes, local debugging bus (Dbus), and the internal computing part. The specific router uses a packet-switched design for node-to-node communication, but a circuit-switched design to connect Dbus and the computing part directly with intracluster core control signals.

The key component in the computing part is a programmable nanoprocessor (***nP***) that is triggered by user-defined debugging events and activates the corresponding debugging processes. The ***nP*** replaces dedicated hardware functions and also handles events by predefined software routines. The particular routines are stored in a small instruction memory within each node. The event handler is composed of an interface and a queue to receive events and stimulate the corresponding behaviors.

The TAP controller (TAPc) is responsible for intracore control. The synchronization module (Sync) is used to manipulate the intercore communications, such as cross-trigger events and synchronizations. However, it is still a challenge for the ***nP*** to provide a quick response or to record frequent events. An alternative solution is to employ a small accelerator (embedded FPGA or configured data paths) for an extended instruction set addition. The configurable accelerator is implemented as a wide communication interface between computing and memory parts to accelerate the debugging processes. In addition, some

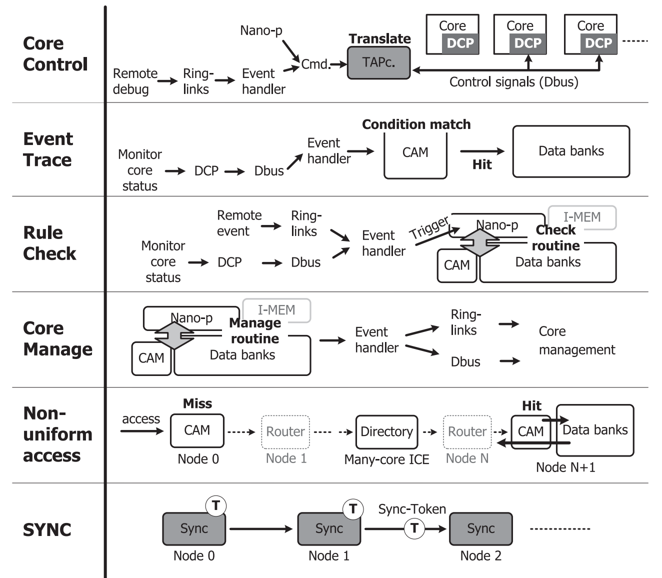


Fig. 4. Functionalities of a NUDA node.

frequent or urgent debugging events should be designed with high priority and short processing latency.

The memory part includes a set of memory banks and CAM in Fig. 3, which satisfy the large capacity and fast search requirements. Moreover, the design strategy of CAM involves a page-based fully associative cache structure to index data banks as shared histogram pages. Distributed shared histogram pages in the NUDA nodes form the proposed non-uniform memory design for runtime debugging facilities, and we present the details in the following section.

Fig. 4 depicts the available functionalities in each NUDA node. Those functions can be combined to perform various debugging facilities, such as variable hardware breakpoints, runtime race detection [11], debugging event order-recording [19], [23], and so on. For a core control instance, the TAPc receives commands from the ***nP*** or the event handler, translates them to sequential control signals on Dbus, and then controls specific cores. When the system is working on runtime debugging facilities, the rule check and event trace functionalities can support system monitoring and event tracing. The CAM can perform rapid condition matching for event tracing, and related events may trigger the ***nP*** to run the rule check routine with the previous histogram stored in the memory of the NUDA node. Moreover, the ***nP*** can monitor the status of the cores and run the core management routine for the deterministic relay of pervious execution or better system performance. The embedding of a nanoprocessor in the NUDA node is demonstrated to provide flexibility and reusability at an affordable cost.

Furthermore, non-uniform debugging memory is organized with non-uniform access for greater scalability, and it creates larger monitor capacity for runtime debugging on many-core systems. The distributed event synchronization mechanism by "Sync-Tokens" helps with global synchronization in debugging. Overall, each NUDA node includes a programmable design implementation by software to satisfy various requirements.

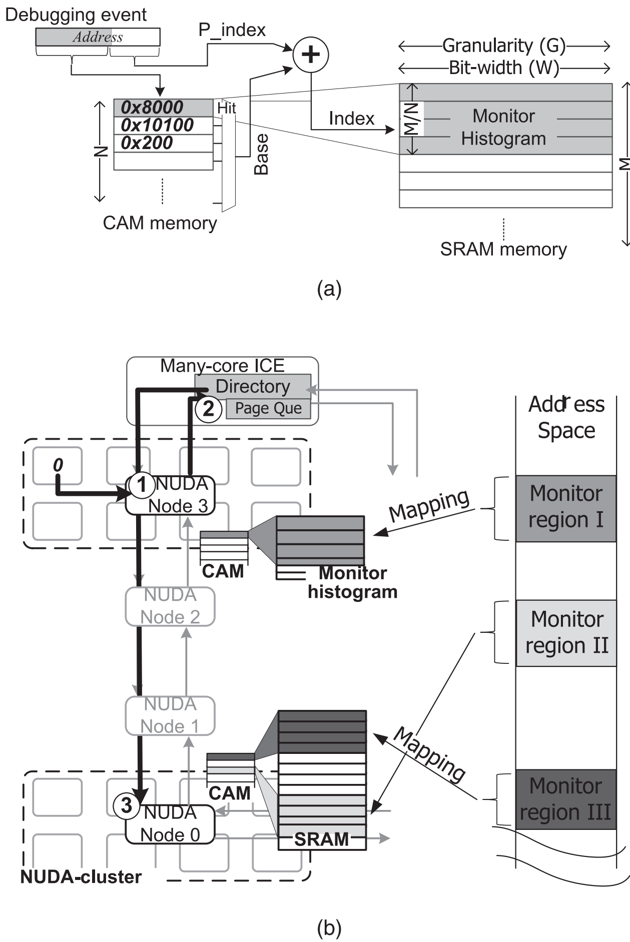


Fig. 5. Non-uniform debugging memory design space. (a) Page-grain cache in a NUDA node. (b) Page-grain NUCA for scalable monitor capacity.

3.2 Non-Uniform Debugging Memory Design Space

The architecture-dependent construction for debugging purposes may suffer from mutual influences between normal and debugging operations. For example, a debugging storage aligned with cache hierarchies must inevitably be polluted by useless debugging events that are used for core execution, and the useful debugging histogram can be lost or swapped out during a cache miss. On the contrary, an independent debugging memory can be well utilized for runtime monitoring without intruding unnecessary events. Therefore, our design strategy use the independent memory for runtime monitoring, but it is distributed, rather than centralized, as a non-uniform cache architecture (NUCA [12]), as shown in Fig. 5b.

The conventional NUCA design features larger shared memory for the benefit of multicore communication, especially in some multithreading programs. It usually has a lower miss-rate and sometimes a lower memory usage. However, it sometimes has the drawbacks of long access latency due to the access distance and ineffective data migrations. In this work, the NUDA contributes a page-based non-uniform memory design for runtime debugging. It gains the benefits of NUCA, of having a larger monitor capacity shared by multiple NUDA nodes without redundant copies. Moreover, the longer latency caused by remote debugging processes only slightly affects the runtime debugging efficiency, for two reasons. First, debugging events happen infrequently. Second, the cores are not blocked while processing debugging events.

Fig. 5a depicts the page-based memory design consisting of CAM and SRAM. The CAM is used for parallel searching for the location of the monitor page, and then accessing the monitor histogram in SRAM. It acts as a fully associative cache to swap out data only when meeting an overflow for minimizing the probability of cache miss.

The page-grain design strategy is not only because of the cost of CAM usage, but also for easy management. Fig. 5b shows the page-grain NUCA distributing in multiple NUDA nodes for scalable monitor capacity. Setting the DCP by user-defined #pragma or compiler aids, interesting address spaces are filtered out, and then monitored by the page-grain NUCA. As shown in Fig. 5b, those interesting regions correspond to a monitor page in a certain NUDA node. For example, interesting region I is mapped to a monitor page in NUDA node 3, and region I/III are mapped to NUDA node 0.

In general programs, access to shared variables is governed by data locality. For example, two threads may access the same shared variables, but at different time slices, by the protected locks, which means that a quantity of debugging events are generated from a thread at a given time slice. If the thread and the monitor page are in the same NUDA cluster, the debugging event triggers an intracluster histogram access. On the other hand, three steps are required for an intercluster histogram access, with longer processing latency. In order to eliminate the global stall caused by too much intercluster histogram access, a good page migration mechanism is necessary, and the information of lock/unlock is a possible reference for prediction.

Fig. 5 (1-3) describes the intercluster debugging histogram access flow. At first, the debugging event accesses the local NUDA node to search for the matching monitor region. If the search fails, the debugging event goes to many-core ICE for a central location directory checking. Then, the many-core ICE passes the event to the remote NUDA node for the intercluster debugging histogram access. Monitor page migration is necessary when different threads are accessing the same shared variables at different time slices. In our design strategy, a page queue in the many-core ICE dynamically records intercluster access counts of current frequent-used pages and makes a decision for migration if the count is over a defined ratio. If a monitor page migration is triggered in a NUDA node with no available storage, it selects a monitor page to replace, and the replaced page is swapped to neighbor NUDA nodes or exchanged with the migration page.

The difference is that with fixed granularity which depends on the cache size [11]. The monitor granularity of NUDA is flexible and user declarable; high granularity G represents a small peephole and precisely monitors the target memory, low granularity represents coarse monitoring, larger monitoring capacity, and fast examination. The relationship between monitor capacity and memory usage is shown in Fig. 6. Essentially, there are two major storage spaces in each NUDA node, the CAM, and the SRAM. The SRAM can be configured whatever the user demands. However, the entry numbers and granularity should follow

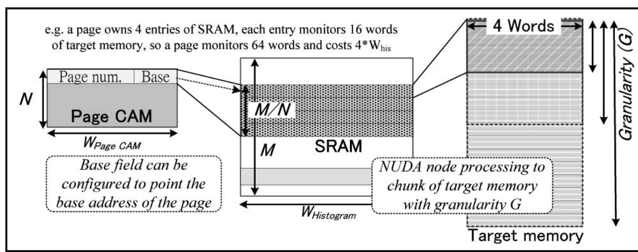


Fig. 6. Monitor granularity, capacity, and memory usage.

the mapping constraint to prevent the memory from going out of bounds.

The proposed non-uniform memory design is flexible to satisfy the requirements of versatile runtime debugging facilities, especially in address-based monitor and its histogram management. For example, it is able to implement runtime race detection and trace for order recording. In addition, unlimited break/watchpoints are supported, by inserting interesting monitor addresses into CAM for parallel comparing.

3.3 Sync-Tokens for Nonintrusive Control

The latency associated with debugging control propagation is challenging. Such latency may interfere with the original execution ordering in a runtime debugging scenario. Nonintrusive control is essential for programmers to fully understand the actual multithreading execution ordering in a system. For example, when a breakpoint is reached by a certain core, all the other cores may stop at different times if there is no synchronized mechanism to bring them to a halt simultaneously. This can cause a programmer to misidentify an actual breakpoint. Our solution is to stop all the cores synchronously and to use a fixed latency to maintain nonintrusive operation. Many-core nonintrusive control plays a key role, as follows:

1. Start/Stop all cores
   - Debugging exception.
   - Tolerate the latency of runtime debugging process.
2. Update all cores
   - Updating NUDA nodes' configuration.
   - Globally examining debugging events.

It is sometimes necessary to start/stop/update all cores simultaneously. While meeting a debugging exception, our strategy relies on synchronously stopping all cores and activating the related handling routine. The debugging exception includes breakpoints, race detection, user-defined assertions alert, histogram overflow, and so on. Sometimes, too many debugging events concurrently happening in close succession or a debugging event which has long processing time may cause the event queue (in DCP or NUDA node) to become full. In order to avoid losing debugging events, our strategy consists of synchronously stopping all cores to digest them as a tolerance without disturbing the original execution order. In other words, it is also necessary to support synchronously updating all NUDA nodes or globally examining debugging events.

In this work, we propose a "Sync-Token" mechanism for synchronous debugging-control. The key idea behind our "Sync-Token" is to pass a specific and related countdown number to each NUDA node, as shown in Fig. 7. This countdown number decreases incrementally with time. When the token passes though all of the NUDA nodes, the countdown numbers across the nodes are allowed to reach zero, whereupon they can be synchronized for start/stop/update operations.

Fig. 7a shows an example of how the sync-token works. Assume that Sync-event A happens in NUDA node N3 and is associated with token number 4. One cycle later, Token A broadcasts to NUDA node N2 and NUDA node N4 and transmits token number 3. At the same time, the token number at NUDA node N3 itself decreases to three. Therefore, NUDA nodes N2, N3, and N4 are synchronized, as shown in Fig. 7b. However, there is another Sync-event B that is being concurrently processed by NUDA node N7. The tokens broadcast conflict, as shown in Fig. 7c. NUDA nodes N1, N2, N3, N4, and N5 are synchronized by token number 2 and NUDA nodes N0, N6, and N7 are synchronized by token number 3. Under our mechanism, lower-numbered tokens can inherit larger numbers, so Sync-event B will hold and wait to be resent at NUDA node N7. When all the tokens across the NUDA nodes count down to zero, all the NUDA nodes can be synchronized for nonintrusive control.

Nonintrusive debugging control is one of the main debugging targets of the NUDA system for greater scalability in future many-core systems. The proposed "Sync-Token" mechanism ensures "synchronously stopping and starting all cores" to tolerate the latency caused overload debugging processes without disturbing the original execution ordering. Indeed, it causes some system performance degradation, but very little, because overload conditions rarely happened.

To deal with a multicore system that runs at more than one clock domain, the NUDA design keeps the main ring and NUDA nodes in the same clock domain. The reason for this choice is better data reference manipulation between NUDA

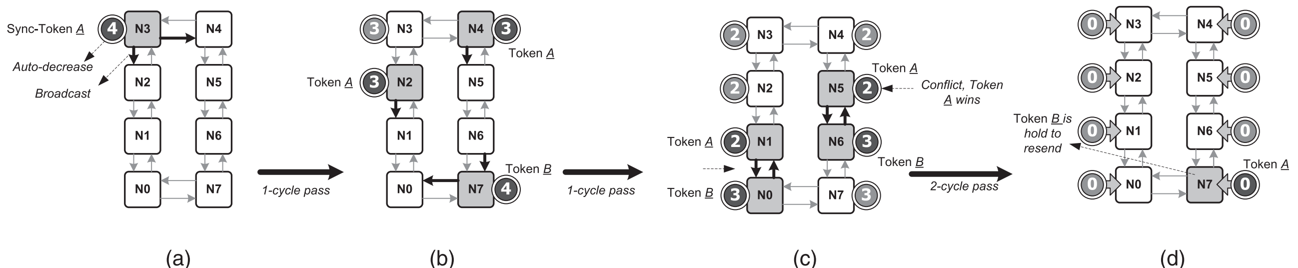


Fig. 7. Example of "Sync-Token" for many-core nonintrusive control. (a) Sync-event $\underline{A}$ happens. (b) Sync-event $\underline{B}$ happens. (c) Conflict, *T_small* inherits *T_large*. (d) Count to zero, start synchronization.

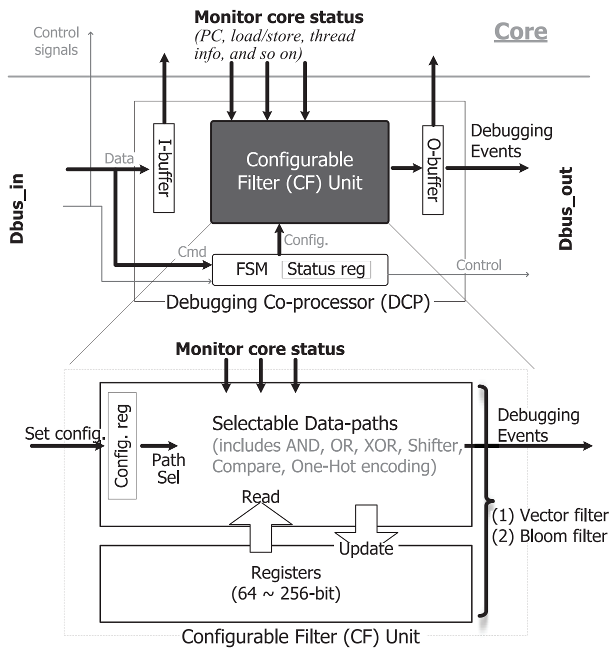


Fig. 8. Configurable filter (CF) unit in DCP.

nodes and an accurate start/stop control on the ring. Therefore, we can classify different clock domains into different NUDA clusters and make sure each NUDA cluster has only one clock domain. Then, we can add an asynchronous module to each NUDA node to handle the different clocks between NUDA node and NUDA cluster. The additional NUDA nodes are the necessary cost of combating different clock domains, but the same clock domain makes modules within the cluster have a simpler design.

3.4 Event Filter and Memory Size Reduction

Debugging events are usually triggered by core/memory/device components, and too many unnecessary events can cause a traffic jam on the NUDA's communication network and degrade the system performance (all cores are stalled when the event queue is full). Moreover, unnecessary events may also intrude in the monitor histogram in the NUDA nodes and lead to low debugging capacity. Consequently, it is crucial to embed an efficient and hardware-feasible event filter in the debugging coprocessor (DCP).

Fig. 8 shows the block diagram of the DCP. Instead of a full in-circuit emulator (ICE) in a single core, the DCP supports the minimal control (control the core and communicate with NUDA) for feasible hardware cost on many-core debugging. The key component, the configurable filter (CF) unit, is used to providing the above mentioned filter functions to improve NUDA architecture efficiency. Our target is to filter out only the necessary debugging events in DCP and send them to the NUDA node. However, precise filtering is unfeasible because it requires many registers and comparators for each distributed debugging events (e.g., the X86 processor only supports four hardware breakpoints). Currently, verifying multiple events by a signature [24] is popular, because the information can be compressed into a fixed length signature without loss to much accuracy. Therefore, using signature for filtering is a good design style for runtime debugging. It can perform at a high filter-rate and reduce the workload on the interconnections. This work provides a configurable filter (CF) unit with two filtering mechanisms for efficiently filtering out most unnecessary events with low cost overhead.

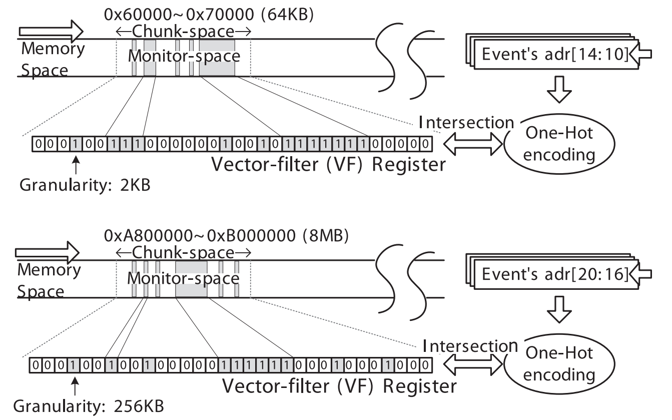


Fig. 9. Example of adaptive rerange "Vector-filter."

Fig. 8 also depicts the composition of the CF unit, which mainly includes a set of registers and selectable data paths. The set of registers can be assumed to be the temporal storage of some candidates that are compressed for filtering, and those registers are totally reusable in the proposed multiple functions. Data paths are built out of multiple selectable paths with several basic operations, including "AND," "OR," "XOR," "Shifter," "Compare," "One-Hot encoding," and so on, by selecting a certain data path for the desired operation and combing registers for reading and updating. We plan to configure two filtering mechanisms ("Vector-filter" and "Bloom-filter"). The cost of the CF unit depends on the size of registers and the operation bandwidth of the data paths.

An adaptive rerange "Vector-filter" is shown in Fig. 9, which features a high filter-rate and low complexity for multiple noncontinuous monitor-spaces. Usually, the interesting regions for monitoring or observing will be user-defined, and those regions may be distributed (different cores or threads in different regions of interest). In the debugging setting, a chunk region is defined for including multiple fragmental monitor spaces. Therefore, there is a rough filter to detect whether or not events hit the chunk region.

Therefore, fragmental monitor spaces can be linked to those small segments by marking specified vector-bits be the mapping transformation. Depending on the range of chunk region, we take out a certain range of the monitor-event's address (e.g., chunk range 64 KB takes the adr[15:10]), and translate it as one-hot code. The one-hot intersects with VF-reg by selectable data paths in the CF unit. If the result is nonempty, this means that the event is likely necessary for debugging. The "Vector-filter" logic is quite simple and low-complexity can easily be embedded in each core. The starting address and range of the chunk region can be adaptively defined by users due to different fragmental monitor spaces. However, if the fragmental monitor spaces are too sparse, the filter rate will be degraded due to the coarser granularity of "Vector-filter."

In order to address the problem of the low filter-rate caused by sparse fragmental monitor spaces, another alternative is to use a "Bloom-filter". As shown in Fig. 10, the basic idea is to compress monitor events into a Bloom-

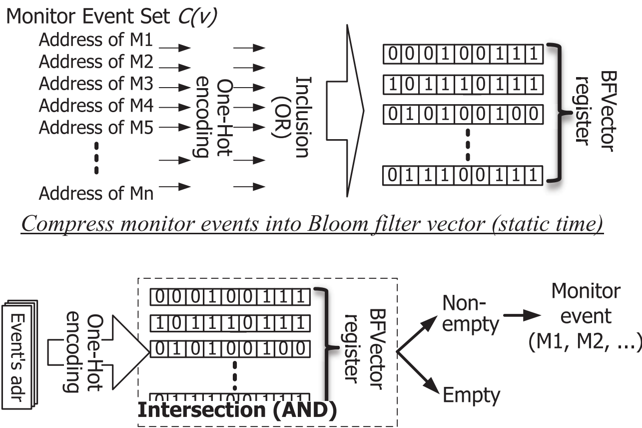


Fig. 10. "Bloom-filter" for sparse fragmental monitor spaces.

```
thread t1,t2,t3,t4;
void race_handler(void* data){
...
}
main_procedure(){
#pragma mdb_configuration global
  {
    SEQ(t1.anchor1, t2.anchor3, t4.anchor6 );
    ESR(race_handler);
    LOCK(t1.S1);
    MONITOR(&Array);
    LOCK(t1.S1 && t4.S2);
    MONITOR(&X);
  }
  t1 = thread_create(…,&some_procedure,...);
  t2 = thread_create(…,&some_procedure,...);
  t4 = thread_create(…,&some_procedure,...);
  return;
}

global char Array[50];
global int X;
#pragma mdb_breakpoint "A"
void some_procedure(void *){
…
#pragma mdb_anchor thread "anchor1"
readlock(&S1);
  #pragma mdb_anchor function "anchor3"
  lock(&S2);
  X++;
  #pragma mdb_anchor thread "anchor6"
  unlock(&S2);
#pragma mdb_breakpoint ##this_position
unreadlock(&S1);
#pragma mdb_watchpoint thread
##this_position
…
}
```

Fig. 11. Code snipped of RunAssert.

filter vector, which is used to filter out the desired monitor events in runtime. The compressed BFVector is stored in the set of registers in the CF unit at a static time. Therefore, in runtime debugging, a monitor event just intersects with the BFVector register to decide whether or not it is a candidate. The method shows a hardware-feasible solution for sparse fragmental monitor spaces, and it has very low miss judgments to allow unnecessary events to pass though. In our experimental results, either of the two filtering mechanisms can perform well, depending on the specific debugging requirements.

3.5 Software Assistance: RunAssert

The NUDA framework includes the software solution for a better experience with parallel program debugging. We used directive #pragma in C language as a programming model, which is composed of directives, macro functions, and assertion expressions in a library. Because it has the characteristics of runtime assertion, we call this tool **RunAssert**. Users can specify the assertion type, range, scope, and target of directives and then define particular rules by macro functions. Finally, assertion expression is an option used to indicate the particular conditions of the debugging rules. The concept of a nonintrusive programming model for parallel program debugging is to decouple the debugging and guarding operations from the ordinary program execution.

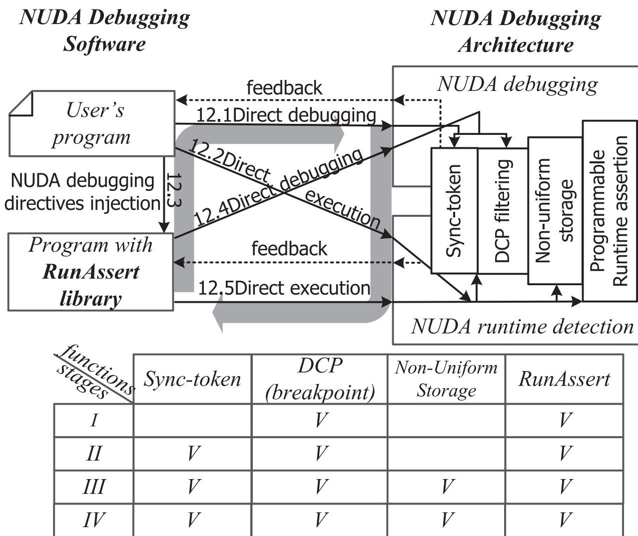
There are two categories of RunAssert directives. The first type is used for parallel program debugging and runtime assertion/guarding. The second type is used for profiling parallel programs. These types of directives can be combined. Moreover, the directives can be separated into in-place and casual types. In-place directives are bound with target source code by source symbols and program counters. The RunAssert compiler links position information to a demand table that records related information, such as program counters and thread symbols. RunAssert helps to debug parallel programs in the following ways.

Acceptable sequence. In developing parallel programs, a programmer's first concern is whether the execution sequence is acceptable. Fig. 11 illustrates the concept of runtime acceptable sequence checking. The #pragrma **mdb_configuration** declares the customized execution sequence, and the #pragma **mdb_anchor** is an anchor inserted into the target program. As a matter of convenience, RunAssert directives can group those anchors into a subgroup, Si. Users can specify the execution sequence by anchors or subgroups. In RunAssert, every function name represents an anchor by default. A useful case is to group the anchors initially by functions and then focus in on the target in particular subgroups. Finally, users use the anchor directive to identify the demand sequence in the suspect region, which was investigated previously.

Race detection. Another case addressed with RunAssert is race detection. Fig. 11 illustrates the race detection flow using RunAssert from source code to reconfigurable logic on NUDA node. The race condition is one of the hardest and most significant problems in parallel programs, as a huge number of debugging events spill from each core into the debugging channel. The key issue in detecting races nonintrusively is how to filter the massive suspected memory accesses. With RunAssert directives, users can easily define race conditions to locate precisely the critical section that requires monitoring. At the beginning, programmer uses the RunAssert directives to identify the target. In this case, we use the directive mdb_lock to indicate the locks and use the directive mdb_shared to indicate the shared objects. Second, the programmer needs to identify the checking rules with objects within the RunAssert global configuration directive. The macro ***ESR(void (*fp)(void*))*** assists users to indicate the exception service routine that is executed as the rules are established. Moreover, the macro ***LOCK*** assists users to identify a guarding lock, and the following macro ***MONITOR*** assists users to list the monitoring shared objects. In order to express the lock hierarchy in the source code, we use the horizontal expression to identify nested locks. In Fig. 3, the shared variable X is under the nested locks S1 and S2, so the RunAssert representative is LOCK(t1.S1 && t4.S2). The main benefit of using the horizontal expression is that users can narrow the monitor scope in the inner locks. For instance, we only need to use LOCK(t1.S2) to describe the process of monitoring S2.

4 A Parallel Program Debugging Paradigm

Although the NUDA architecture provides a fast and nonintrusive parallel program debugging environment, it is not straightforward or easy to do so. Actually, in the process of software development, debugging occupies most of the time [25], [26]. In fact, the total ordering characteristic of parallel programs implies that detecting and fixing the partial ordering errors cannot guarantee the total ordering



functions / stages	Sync-token	DCP (breakpoint)	Non-Uniform Storage	RunAssert
I		V		V
II	V	V		V
III	V	V	V	V
IV	V	V	V	V

Fig. 12. The proposed parallel program debugging paradigm.

correctness. The hardest part in debugging parallel programs, or even the traditional sequential programs, is identifying the bugs. Currently; the interactive debugging is the most famous and important methodology to identify faults in the program. However, it is unrealistic to identify faults step by step, especially when the user is faced with a large and complex program. For this reason, this work proposed a parallel program debugging paradigm. The high speed automatic bug detection can help users to locate probable bug locations as quickly as the program runs under normal execution speed, and the high precision interactive debugging can point out the bugs and solve them efficiently.

Fig. 12 illustrates the novel parallel program debugging cycle proposed in this work. According to RunAssert and the NUDA debugging architecture, programmer can trace the faults by iterations.

The NUDA debugging architecture is composed of NUDA runtime detection and a NUDA direct debugging interface. Both of them share four key features of NUDA, a programmable user-defined function unit for RunAssert, synch-tokens for the synchronized triggering of global debugging events, DCP filters for high speed and high performance debugging event filtering, and non-uniform storage to provide a larger and smarter storage space for particular histogram recording.

The interaction cycle between debugging software and NUDA architectures can be illustrated in Fig. 12. A waterfall debugging flow based on the NUDA debugging paradigm can be further elaborated in the following four steps with debugging functionalilities we used.

1) General sequential debug. Programmers develop their routines in a normal sequential circumstance. The traditional debugging tools, such as gdb, perform very well in this stage. However, NUDA's DCP filtering can leverage the legacy debugging tools in the manycore environment (Fig. 12.1). Moreover, the RunAssert can be inserted into the target program as an on-the-fly debugger (Fig. 12.3). Because the NUDA has an online monitoring property, the RunAssert can help to monitor the logical faults without influencing performance, and even better, the RunAssert code can remain in the program to provide human readable debugging information for future maintenance (Figs. 12.2 and 12.4).

2) General parallel debug. As the components of those routines are run in parallel, programmers usually need a parallel program debugger to deal with the synchronous "capture-and-stop" mechanism. The proposed sync-token mechanism in NUDA can help the many-core system stop/resume precisely and simultaneously according by user specifying (Figs. 12.1 and 12.4). More hardware breakpoints/watchpoints are supported to enhance the parallel debug capability.

3) Parallel program behavior debug. After most logical faults are detected, the parallel program development is moved on overall detection stage to identify parallel faults. RunAssert supports to check the program execution sequence, coupled with high-level languages, such as C language. Programmers can define the check points, called anchors, by RunAssert, and then define an expected execution sequence with flexible RunAssert descriptions. At runtime, NUDA will perform the nonintrusive checking in order to ensure that the program behavior is executed as specified (Fig. 12.5).

4) Runtime race detection. In case of suspicious race conditions existing in a program, programmers can narrow down the focus of the certain detection range by only monitoring activities within specific memory range. In general, larger ranges result into higher false positive rate. However, with less number of comparisons, the NUDA dealing with a larger monitor range is faster than the one with a smaller range. By iteratively refining monitor ranges, programmers can gradually "zoom in" those problems in the suspected memory area.

Whenever to catch a fault in each of the above four steps, programmers can go back to the previous steps smoothly or remaining in the current step to fix the faults with RunAssert and NUDAs' support.

5 NUDA Application

This section will introduce the automatically monitoring capability of NUDA system. We will demonstrate NUDA under different configurations to handle data race detection and deadlock detection. In addition, several advanced design issues are discussed, including histogram migration for less remote race detection, overflow handling, and more efficient barrier handling.

5.1 Data Race Detection Flow

This work proposes a case study of hardware lockset-based race detection by configuring a NUDA. Our approach is based on the previous work [11], which uses hardware Bloom filters, but features false negative elimination, high scalability, and nonintrusion via the independent NUDA structure and its efficient histogram management strategy.

The Eraser [15] is one of the most popular data race detection algorithm. The key components of Eraser are as following:

1. Thread lockset L(t): locks currently held by a thread.
2. Candidate set C(v): locks for variables protection.
3. Access state "LState": pruning false positive.

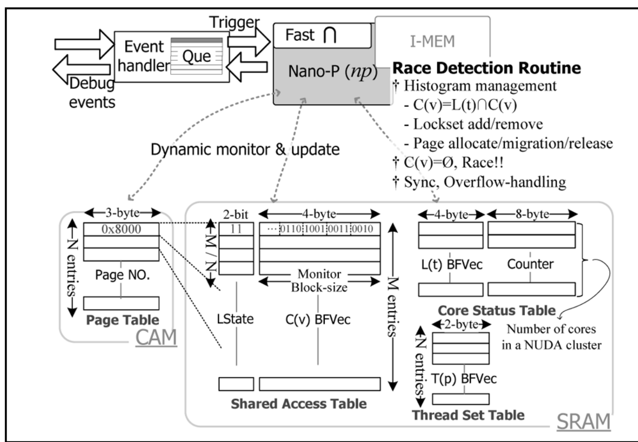


Fig. 13. The race detection configuration in a NUDA node.

The HARD [11] translates C(v) and L(t) into fixed-length Bloom filter vectors and implements them on traditional memory hierarchies for the first approach to hardware-assisted lockset-based race detection.

Fig. 13 shows the configurations of memory and race detection processing in a NUDA node. To maintain data structures in Eraser [15] and HARD [11], four tables are configured for the monitor histogram by using the internal memory built into the NUDA node. There are three tables in SRAM memory, and one table in CAM for fast comparing or searching. At first, the main histogram table (called the "Shared Access Table") in SRAM records the access status (LState) and compressed locks (C(v) BFVector) of each variable. The required monitor capacity can be adaptively configured by the defined monitor granularity, bit-width of C(v) BFVector, and how many entries in "Shared Access Table." Next, "Page Table" is used to index "Shared Access Table" by dividing it into different monitor pages, and it is constructed in CAM to support the fast searching of page locations, and to reduce the probability of data swap-out. Third, "Core Status Table" in SRAM memory is responsible for maintaining those locksets (L(t) BFVector) and Counter registers (for BFVector adding/removing locks) currently held by the threads. The number of "Core Status Table" entries is equal to the number of cores in the NUDA cluster. Finally, "Thread Set Table" is used for our barrier handling mechanism to address advanced design issues, and it records the thread access histogram on each monitor page.

For the configuration example shown in Fig. 13, a 64-core system with 8 NUDA clusters requires 128 KB monitor capacity for shared variables and 32 B monitor granularity. So each NUDA cluster is responsible for 16 KB monitor capacity, and BFVector is extended to 4B for rare collisions. If a monitor page size is defined as 1 KB, then "Page Table" will contain 16 entries ($\sim$3B per entry for page number), and there are 512 entries (16 KB/32 B) and $\sim$4B per entry in "Shared Access Table." "Core Status Table" has eight entries corresponding to the number of cores in a NUDA cluster, and it costs 12 B $(4 + 8)$ per entry to record L(t) and Counter registers. "Thread Set Table" has the same entries as "Page Table" to record the thread access histogram by T(p) BFVector (2 B). In this example, the total memory usage of a NUDA node costs about

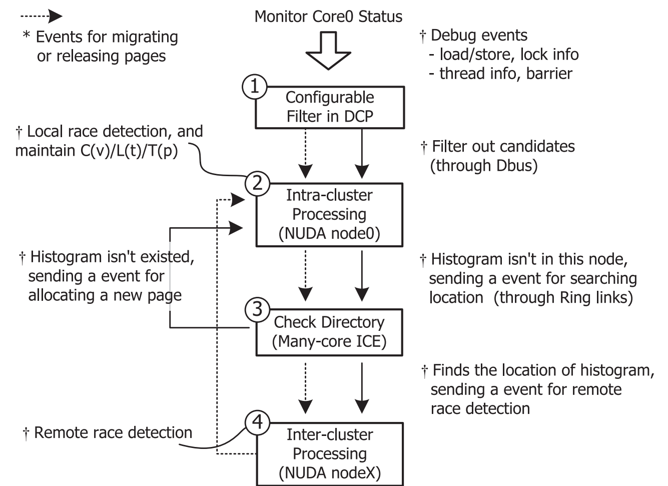


Fig. 14. Proposed race detection flow on NUDA.

2.2 KB $(512 * 4\text{B} + 8 * 12\text{B} + 16 * 2\text{B})$ SRAM memory and 48 B $(16 * 3\text{B})$ CAM.

In race detection processing, the event handler takes care of receiving and sending debugging events and triggers a nanoprocessor (***np***) to execute the race detection routine. The ***np*** built into the NUDA node is programmable to support several functions, not only detecting race conditions. The race detection routine includes histogram management, race detection, event synchronization, and exception handling (like overflow) for flow completion. The ***np*** processing speed should be considered not to slow down the whole debugging system, and the configured data paths in ***np*** can accelerate it with specific instructions (like fast intersection in this case study). In order to improve the processing speed of ***np***, the most frequent operation "race detection, then update C(v)" should be accelerated. In this work, the ***np*** design with accelerated (configurable) data paths can process this operation in five cycles (event trigger $\rightarrow$ searching page location $\rightarrow$ parallel read C(v) and L(t) $\rightarrow$ intersection C(v) and L(t) and if empty, possible race $\rightarrow$ update C(v)). The other operations, like L(t)/monitor page/overflow managements, are not critical (they happen infrequently) and are designed to have a longer processing latency.

After illustrating the lockset-based race detection configuration in a single NUDA node, Fig. 14 represents the overall race detection flow on NUDA, and there are four processing steps at most. The solid line illustrates the non-uniform race detection, and the histogram management represents in the dotted line. In the first step, debugging events (load/store, locks, and so on) will be filtered out by the configurable filter in DCP and sent to the related NUDA node through the local debugging bus (Dbus). While the debugging events reach the related NUDA node, intracluster processing is triggered as the next step, and the process may contain local race detection or a histogram update according to the event type. In our design strategy, the non-uniform memory for the shared variables histogram (C(v)) is distributed in and shared for each NUDA node. If the monitored shared variable is not in the local NUDA node, remote race detection is triggered in our third step. Through ring-links between nodes, the remote race detection looks up the directory in many-core ICE for searching for the

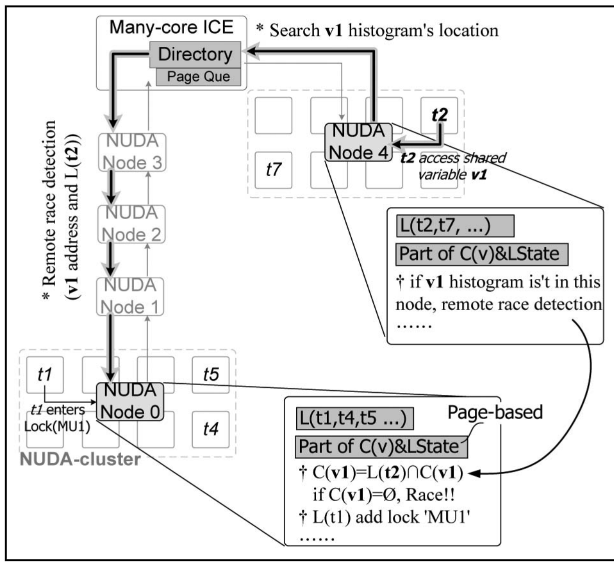


Fig. 15. Example of non-uniform race detection on NUDA.

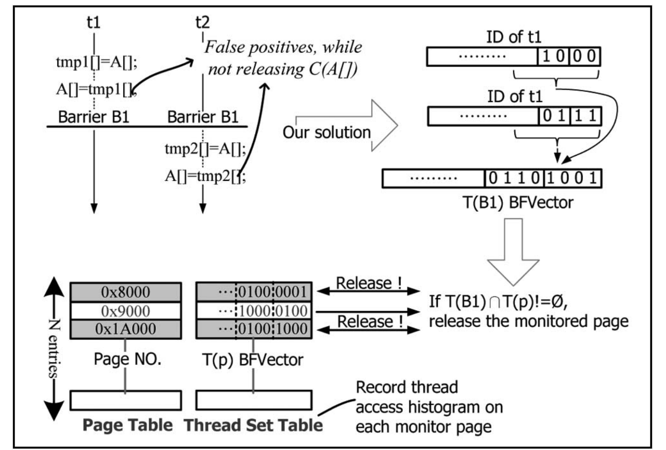


Fig. 16. Our barrier handling mechanism for releasing partial monitor pages.

variable's location. If the location is existed, the many-core ICE passes the debugging event to the related NUDA node for remote race detection. On the other hand, if the location does not exist, the many-core ICE sends a management event back to the original requested NUDA node for new monitor page allocation.

In the other monitor histogram management, the many-core ICE detects monitor pages that are frequently used in remote race detection and migrating those pages to related local NUDA nodes for eliminating overall race detection latency. Releasing monitor pages is another important management to avoid data overflow, and we discuss those advanced design issues in the next section.

Fig. 15 shows an example of non-uniform race detection. There are several NUDA clusters with their NUDA nodes, and we present two of them in detail as an example. Multiple threads are executed in parallel in different cores and different NUDA clusters, and their NUDA nodes dynamically monitor debugging events, check violations, and update histograms. In this example, thread 1 (t1) enters a lock (MU1) at the current running program, and the lock event triggers "NUDA node0" to update L(t1) with adding a new lock (MU1). Concurrently, a shared access (v1) by thread 2 (t2) is filtered out and sent to "NUDA node4" for local race detection. However, the v1 histogram is not in "NUDA node4," and remote race detection is triggered, as shown in Fig. 12. At first, the remote race detection process uses the address of v1 to search for its location in the many-core ICE directory. In this case, the histogram C(v1) is in "NUDA node 0," and the v1 address and t2 lockset L(t1) are sent to the node for completing the race detection ($\mathrm{C(v1) = L(t2) \cap C(v1)}$. If C(v1) is empty, there is a possible race alert).

5.2 Advanced Design Issues

In this section, we discuss several advanced design issues with the NUDA, to improve the many-core runtime race detection efficiency. Those design issues are targeted toward low false positive warnings, catching possible race conditions, low system performance overhead, and the backend analysis.

5.2.1 Debugging Event Filter

Too many unnecessary debugging events (not in monitor space) will cause a traffic jam on the NUDA network to degrade the system performance. In addition, those events also intrude histogram-tables into the NUDA nodes and lead to low monitor capacity. For race detection, the filter should only filter out the desired shared variables. Since shared variables are distributed in memory space, it is vary hard to use small hardware to exactly filter them. Our design strategy is to gather possible or desired shared variables and compressing their addresses (page numbers) into a Bloom filter vector, and then configuring the vector into the proposed filter in DCP. Therefore, all desired shared variables can be filtered out while only mixing up very few undesired variables due to the Bloom filter collisions, and our experimental result shows its efficiency.

5.2.2 Barrier Handling

As shown in Fig. 16, barriers cause many false positives in the lockset algorithm, because the accesses from different threads to a variable can be ordered by barriers without races. HARD [11] observed the problem but does not have a complete solution, and its barrier handling mechanism is to discard all histograms in the cache hierarchy. However, if different groups of threads go through different barriers, or some threads do not go through the barrier, this approach may cause false negatives. Our barrier handling mechanism discards partial histograms by checking the "Thread Set Table" and avoids the negative effect.

Fig. 16 represents an example of releasing partial monitor pages. Thread 1 (t1) accessed "Array A" before "Barrier B1," and thread 2 (t2) accessed "Array A" after "Barrier B1." "Thread Set Table" records the thread access histogram in the form of a BFVector on each monitor page. We also modify the barrier API in the thread library to launch a page releasing command to the NUDA to release the related monitor pages. In this example, the barrier API gathers the IDs of the threads that are grouped to go through "Barrier B1" to produce T(B1) BFVector and sets the DCP as a page releasing command. Then, the T(B1)

TABLE 1
Target System Parameters

Simulator parameters				
Simulator	I	mcore	ISA	x86
	#core	64	CPU family	Intel® Atom™
	II	Intel® PIN	ISA	X86-64
	#core	32	CPU family	AMD® Opteron™

NUDA parameters	
#cores in cluster	4, 8
parallel processing ability in NUDA-node	single, bi-direction
bit-width of ring interconnection (bit)	8, 16
latency of ring interconnection (cycle)	1, 2
DCP buffer size (word)	4,8
NUDA-node monitor block size (words)	8, 16, 32
Shared access table size	16~256KB
CAM mapped page size	0.5~4KB

BFVector s broadcasted to each NUDA node, and the np uses it to check each entry of "Thread Set Table." If T(B1) BFVector ∩ T(p) BFVector is not empty, it means that thread 1 or 2 had accessed the monitor page, and then released it.

5.2.3 Avoid Monitor Histogram Overflow

Given the limited hardware monitor capacity, it is important to release monitor pages in order to avoid histogram overflow. In the lockset algorithm, going through a barrier is the only way to release them. Fortunately, barriers are commonly used in multithreading programs, and our barrier handling mechanism (Fig. 16) can precisely release useless monitor pages without causing both false positives and negatives. In addition, the good programming concept of using global barriers (programming complexity reduction and reliability improvement) should be promoted both in compiler techniques and in user programming guides, and it also helps for the NUDA to have a feasible monitor capacity. When a histogram overflow occurs, the many-core ICE selects a monitor page and swaps it out, and notifies the programmers of the overflow information.

6 EXPERIMENTS

In this work, we contribute two different simulation environments for the NUDA evaluation. Table 1 shows a parallel simulator mcore [27] in the context of the SPLASH2 benchmarks [28]. In order to support more benchmarks, we also use Intel PIN for this purpose. PIN is a dynamic instrumentation tool that allows users to contribute their PIN tools for different purposes. We used seven race-free

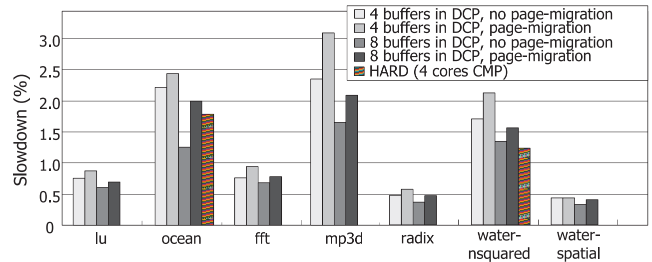


Fig. 17. Slowdown effects.

SPLASH2 benchmarks to evaluate the NUDA. Those benchmarks work with a lock-based multithreading programming model.

Race detection by the NUDA is essentially real-time and nonintrusive to the original execution. However, in the case that the monitoring buffer is full, the whole system has to be stopped. Table 2 illustrates the NUDA overhead in SPLASH-2 for runtime race detection. We found the overhead to be inversely proportional to the buffer size; more DCP buffers can tolerate more concurrent memory access events. Fig. 17 shows the slowdown effects with respect to buffer size; the average overhead for 4-word to 8-word buffers is small. In addition, the stall caused by shared access table migration is also low. This is because most of the benchmarks use the shared spaces separately. In contrast, we note that benchmarks ocean, mp3d, and water-nsquared all have high density spaces and higher migration stall rates. The page usage column in Table 2 shows the maximum usage of the shared access table pages across the whole system. Lower page usage rates can reduce intercluster traffic and improve performance. The inter-/intracluster ratio represents the relation between intercluster and intracluster events. More intercluster events cause more chances for NUDA migration and affect the performance.

Table 3 shows the results in comparison with other approaches, the race detection. The hardware approach, [11], offers fewer false positives than Valgrind and operates as fast as runtime (0.1-2.6% ↓). However, it depends on the cache and cache coherence mechanisms. We believe that this implicit cost of cache coherence is too high and is infeasible in most many-core environments. Our proposed NUDA features nonintrusive truly race detection without false positives. The system offers negligible slowdown (0.51-23.06% ↓), and supports user defined assertions.

TABLE 2
Benchmarks' Statistics, Resource Requirements, Debugging Event Rate, and Performance

Benchamarks	Data set	Shared Data size	Max. page usage	Shared access events	Locks events (number/10^6cycles)	Inter-/Intra-cluster	Slowdown	Buffer-full stall	Migration stall
lu	512 x 512	132KB	41	10.67%	0.002144	0.04% / 99.6%	0.814%	0.753%	0.061%
ocean	130 x 130	132KB	63	1.8%	432.2	19.3% / 80.7%	2.5%	0.525%	0.975%
fft	256K	396KB	52	3.3%	0.00322	64.9% / 35.1%	0.93%	0.068%	0.25%
mp3d	-	132KB	64	2.66%	0.000133	2.7% / 97.3%	3.06%	2.24%	0.82%
radix	4M keys	396KB	33	9.4%	0.00543	1.2% / 98.8%	0.51%	0.486%	0.024%
water-nsqared	512	132KB	61	4%	0.00771	49.2% / 50.8%	2.2%	1.82%	0.38%
water-spatial	512	132KB	40	1.2%	0.00802	37.56% / 62.44%	0.422%	0.225%	0.197%

* Rate of Shared access events = #shared memory access/#total memory access, Inter-/Intra-cluster, Slowdown, Buffer-full stall, Migration stall = cycles / total execution cycles

TABLE 3
Comparisons of Race Detection Methods (64-Core)

	Valgrind* [8]	HARD† [11]	NUDA‡ (This Work)
Slowdown	60-400X	0.1-2.6%	0.1-3.06%
Non-intrusive	No	Probably Yes	Yes
False positive/negative	Positive	Both	None
User-defined assertions	No	No	Yes
Cache coherency	No	Required	No
Memory-bit usage§	No	64KB extra in L1$, 1MB extra in L2$	64KB extra SRAM, 1KB extra CAM
Area estimation¶ (65nm)	0.0%	$5.53mm^2$ (0.98%\\)	$2.08mm^2$ (0.37%\\)

*** Valgrind works on Intel Core2 Quad CPU 2.5G RAM 2G workstation with Linux kernel 2.6.24**

† **The SPEC of HARD [11] (normalize to 65nm):** 2B BFVector/L1 $line ($1.94mm^2$), 2B BFVector/L2 $line ($3.58mm^2$), the estimated area: **5.53** mm^2.

‡ **The proposed SPEC of NUDA (normalize to 65nm):** 8 NUDA clusters, 8KB SRAM ($8*0.19mm^2$), 32entries CAM ($8*0.041mm^2$) in each NUDA node (C:32/P:1KB/B:32B), 2B-width local *D*bus in each NUDA cluster ($0.1mm^2$), ring interconnection (9 nodes, 2 rings, 1B-width/per-ring) ($0.135mm^2$), the estimated area: **2.08** mm^2.

§ **Memory ref (cacti 5.3v, 65nm):** 32B/line 2048-entry DRAM: $0.34mm^2$, 2B/line 2048-entry DRAM: $0.014mm^2$, 2B/line 128-entry SRAM: $0.0076mm^2$, 8B/line 1024-entry SRAM: $0.19mm^2$, 4B/line 32-entry CAM: $0.041mm^2$

¶ **The assumed SPEC of many-core SOC (normalize to 65nm):** 64 Atom cores with 16KB 4-way 32B/line I/D-cache ($6.1mm^2$ per core), 16MB 8-way 32B/line 256 banks L2-NUCA with Mesh NoC ($174.4mm^2$), the estimated area: **564** mm^2.

\\ **Many-core chip ref:** Intel's 80-Core (80 tiles, 65nm): $275mm^2$, $0.5mm^2$ (per router), The CELL processor (12 tiles, 90nm): $235mm^2$, The UltraSPARC T1 (14 tiles, 90nmm,): $378mm^2$,The Atom (1 tiles, 45nm): $25mm^2$, EIB in CELL (12 nodes, 4 rings, 16B-width/per-ring, 90nm): $5.98mm^2$.

TABLE 4
Race Detection by Renaming Locks

Benchmarks	Faults info. inject	Faults info. appear	Valgrind [30](slowdown)	NUDA
lu	68	2584	48688 (36)	118
ocean	208	5148	128953(34)	354
fft	38	365	258129(74)	98
mp3d	102	4025	-	225
radix	414	11642	535020(36)	454
water-nsqared	19	524	15375(27)	55
water-spatial	20	601	6159(28)	34

Table 3 also shows the related hardware cost estimates for several popular many-core processors, interconnection units, and memory, by CACTI 5.3 [29]. We assume that the many-core environment is composed of 64 Intel Atom processors, with 16 KB I/D cache for each core, and 16 MB L2-NUCA for sharing. The total estimated area is 564 mm2 under 65 nm technology. Comparing HARD and NUDA, the main factors in our estimates were memory-usage and interconnections. Other elements (logic, FSM, etc.) do not impact the chip area. In HARD, the BFvector (2 B/per line) in the L1 cache is estimated from SRAM usage, while the BFvector (2 B/per line) in the L2 cache is estimated from DRAM usage. There are eight NUDA nodes in our proposed system. In addition, the width of the ring interconnection that links all nine nodes (include the many-core ICE) is 1 B, and we also include the ring routers. As shown in Table 3, the HARD has 0.98% (5.53/564) of the area cost compared with the proposed many-core system, and this work has 0.37% (2.08/564) overhead.

Table 4 compares the detected races between the NUDA and the software race detection tool, Valgrind. We modified the SPLASH2 macro to create "delta locks," where the library on purpose acquires/releases the same lock but using different locks in each thread. By the delta lock, the parallel program apparently produced incorrect results, because of many race conditions. The first column provides the number of faults we injected and the possible fault points that may be found in the parallel program. Because we injected the fault in a static program sequence, those faults expanded during runtime. For example, a single delta lock in source code can derive multiple potential fault points due to thread parallelism. The third and forth columns show the faults caught by NUDA and Valgrind [30]. This work was not meant to be a direct comparison with HARD because HARD cannot be modeled precisely in a 64-core system. However, according to the literature of HARD, false positives and false negatives would be significantly increased because of insufficient Bloom filter size and L1 cache misses.

There are two important observations as shown in Table 4. First, the NUDA is faster than the software solution. Basically, the Valgrind is 30 times slower than the normal execution on average. Second, the nonunified memory supports more storage space for the memory access histogram. NUDA can support higher precision than Valgrind. In fact, it is not impossible for Valgrind to reach the same precision, but the race detection time will become unacceptable.

7 Conclusion

The main theme of this paper is to contribute a nonintrusive debugging framework for many-core systems. The key features include the fact that 1) the NUDA is operated in parallel to the original data interconnection, enabling "nonintrusive" debugging methods. Complicated debugging methodologies for parallel programs can be supported without causing a probe effect or behavior distortion, 2) A cluster-based ring interconnection facilitates local communication exploration and the related Sync-Token protocol on the ring guarantee the accuracy of debugging operations, and 3) The NUDA-node (consisting of memory, configuration logic, and a nanoprocessor) provides a unified structure to support advanced debugging with minimal hardware cost. Finally, we demonstrate the implementation of three published testing and debugging schemes on our NUDA architecture.

References

[1] C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs," *J. ACM Computing Surveys,* vol. 21, no. 4, pp. 593-622, 1989.

[2] L. Seiler et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Trans. Graphics,* vol. 27, no. 3, pp. 1-15, 2008.

[3] nVidia, "Next Generation CUDA Architecture," http://www.nvidia.com/object/fermi_architecture.html, 2011.

[4] B. Vermeulen, M.Z. Urfianto, and S.K. Goel, "Automatic Generation of Breakpoint Hardware for Silicon Debug," *Proc. 41st Ann. Design Automation Conf.,* 2004.

[5] K. Goossens et al., "Transaction-Based Communication-Centric Debug," *Proc. First Int'l Symp. Networks-on-Chip,* 2007.

[6] "Standard Debug Interface Socket Requirements for OCP-Compliant SoC."

[7] S. Tang and Q. Xu, "In-Band Cross-Trigger Event Transmission for Transaction-Based Debug," *Proc. Conf. Design, Automation and Test in Europe,* 2008.

[8] A.R.M. Ltd., "CoreSight Architecture Specification," 2004.

[9] R. Leatherman, "On-Chip Instrumentation Approach to System-on-Chip Development," *OCI White Paper,* available at http://www.fs2.com, 2011.

[10] S. Min and J. Choi, "An Efficient Cache-Based Access Anomaly Detection Scheme," *ACM SIGARCH Computer Architecture News,* vol. 19, no. 2, pp. 235-244, 1991.

[11] P. Zhou, R. Teodorescu, and Y. Zhou, "HARD: Hardware-Assisted Lockset-Based Race Detection," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture*, 2007.

[12] J. Huh et al., "A NUCA Substrate for Flexible CMP Cache Sharing," *Proc. 19th Ann. Int'l Conf. Supercomputing*, 2005.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[14] P. Keleher, A. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Distributed Shared Memory: Concepts and Systems*, p. 96, 1998.

[15] S. Savage et al., "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems*, vol. 15, no. 4, pp. 391-411, 1997.

[16] J.W. Voung, R. Jhala, and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code," *Proc. Sixth Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC-FSE '07)*, 2007.

[17] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," *ACM SIGOPS Operating Systems Rev.*, vol. 39, no. 5, pp. 221-234, 2005.

[18] M. Singhal and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47-52, 1992.

[19] M. Xu, R. Bodik, and M.D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," *Proc. 30th Ann. Int'l Symp. Computer Architecture*, 2003.

[20] D.R. Hower and M.D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," *Proc. 35th Ann. Int'l Symp. Computer Architecture*, 2008.

[21] A. Alameldeen et al., "Evaluating Non-Deterministic Multi-Threaded Commercial Workloads," *Proc. Fifth Workshop Computer Architecture Evaluation Using Commercial Workloads*, pp. 30-38, 2002.

[22] R. Chandra et al., *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, Inc., 2001.

[23] M. Ronsse and K.D. Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System," *ACM Trans. Computer Systems*, vol. 17, no. 2, pp. 133-152, 1999.

[24] A. Muzahid et al., "SigRace: Signature-Based Data Race Detection," *Proc. 36th Ann. Int'l Symp. Computer Architecture*, 2009.

[25] B. Boehm, "Software and Its Impact: A Quantitative Assessment," *Software Eng.: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, vol. 19, no. 5, p. 91, 2007.

[26] B. Boehm, "Improving Software Productivity," *Software Eng.: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, vol. 20, no. 9, p. 151, 2007.

[27] A.-T. Nguyen et al., "The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures," *Proc. Int'l Conf. Computer Design, VLSI in Computers and Processors*, 1996.

[28] S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, 1995.

[29] S. Wilton and N. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, pp. 677-688, May 1996.

[30] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2007.

[31] M.-C. Hsieh and C.-T. Huang, "An Embedded Infrastructure of Debug and Trace Interface for the DSP Platform," *Proc. 45th Ann. Design Automation Conf.*, 2008.

Chi-Neng Wen received the MS degree from computer science and information engineering department at the National Chung Cheng University, Chia-Yi, Taiwan, in 2006, and is currently working toward the PhD degree in the same affiliation. His research interests include Electronic System Level simulation and verification methodologies, embedded system hardware/software codesign, multicore architecture design, and multicore debugging from improving the real-time race detection and system monitoring performance.

Shu-Hsuan Chou received the BS and master's degrees from the Department of Computer Science and Information Engineering at National Chung Cheng University (CCU), Chia-Yi, Taiwan, in 2004 and 2005, respectively. He is currently working toward the PhD degree in Department of Computer Science and Information Engineering at National Chung Cheng University. His recent research has produced multithreading/multicore media processors, multicore on-chip networks/memory system, multicore debugging architecture, and adaptive processor architecture techniques. His current research interests include multicore SOC design, embedded system design, low power methodology, and design for characterization.

Chien-Chih Chen received the BS and the MS degrees in Department of Computer Science and Information Engineering from National Chung Cheng University (CCU), Chia-Yi, Taiwan, in 2007 and 2009, respectively. He is currently working toward the PhD degree in Department of Computer Science at National Chung Cheng University. His research interests include multicore SOC design, embedded system design, and computer architecture.

Tien-Fu Chen received the BS degree in computer science from National Taiwan University (NTU), Taipei, Taiwan, in 1983. He received the MS and PhD degrees in computer science and engineering from the University of Washington, Washington D.C., in 1991 and 1993, respectively. He joined Wang Computer Ltd., Taiwan, where he worked as a System Software Engineer for three years. Currently, he is a professor with the Department of Computer Science and Information Engineering, National Chiao Tung University (NCTU), Taiwan. He has published several widely-cited papers on dynamic hardware prefetching algorithms and designs. He has made contributions to processor design and system-on-chip (SoC) design methodology. His recent research has produced multithreading/multicore media processors, on-chip networks, and low-power architecture techniques, as well as related software support tools and SoC design environments. His current research interests include computer architectures, SoC design, and embedded systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.