Quantifying the Potential of Program Analysis Peripherals

Mohit Tiwari, Shashidhar Mysore, Timothy Sherwood
Department of Computer Science
University of California, Santa Barbara
tiwari,shashimc,sherwood@cs.ucsb.edu

***Abstract*—As programmers are asked to manage more complicated parallel machines, it is likely that they will become increasingly dependent on tools such as multi-threaded data race detectors, memory bounds checkers, dynamic dataflow trackers, and various performance profilers to understand and maintain their software. As these tools continue to grow in importance, it is worth exploring the potential for special purpose accelerators for these tasks, especially since commodity multi-cores can only provide limited speedups. Rather than performing all the instrumentation and analysis on the main processor, we explore the idea of using the increasingly high-throughput board level interconnect available on many systems to offload analysis to a parallel off-chip accelerator. There are many non-trivial technical issues in taking such an approach that may not appear in simulation, and to flush them out we have developed a prototype system that maps a DMA based analysis engine, sitting on a PCI-mounted FPGA, into the Valgrind instrumentation framework. Using this prototype we characterize the potential of such a system to both accelerate existing software development tools and enable a new class of heavyweight dynamic analysis. While many issues still remain with such an approach, we demonstrate that program analysis speedups of 29% to 440% could be achieved today with strictly off-the-shelf components on some of the state-of-the-art tools, and we carefully quantify the bottlenecks to illuminate several new opportunities for further architectural innovation.**

***Keywords*-Dynamic program analysis, Hardware support, Debugging and Security, Offchip accelerator**

I. Introduction

Developing fast, quality software on a modern computer system is by no means easy. Even today, software bugs are so damaging and widespread that they cost the U.S. economy alone an estimated $59.5 billion annually (more than half a percent of the US GNP). Although it is certainly not possible to remove all errors, it is estimated that more than a third of the cost associated with bugs could be eliminated through an improved testing and analysis infrastructure [22]. The problems associated with inefficient and buggy software are not going to be helped by the fact that the amount of hardware complexity exposed to the programmer is growing rapidly on desktop and server machines in the form of threading, parallelism, and complex application middleware. To cope with this complexity, and ensure the quality of software infrastructure, an increased reliance on sophisticated software analysis and testing tools seems inevitable.

One important aspect of the testing problem is that complex pointer errors, memory leaks, race conditions, and performance anomalies may manifest themselves during tests, but finding them requires sifting through a sea of runtime data. To find these errors, many people rely on dynamic analysis frameworks such as Valgrind [19], Pin [14], and DynamoRIO [1]. While dynamic analysis can be done completely in software through binary instrumentation, the amount of analysis that can be done at test-time is bounded by the performance impact that can be tolerated. The performance impact is especially critical in long running and interactive programs (keep in mind even loading a page in Firefox takes about 900 million instructions), or when analysis methods require significant computation be performed on-line with the execution (for example in tracking all the memory addresses in a race condition detector).

In this paper, we explore the potential for off-chip hardware-accelerated software analysis tools that can sift through on-line profile data. In particular, we consider a class of designs where the profiling support is parallelized between the on-chip task of data gathering and the off-chip task of actual analysis. The on-chip system, which may even be implemented strictly in software, gathers and packs data for transport. The off-chip system then unpacks the data and performs the actual dynamic program analysis. To uncover the potential issues with such an approach, for this paper we have designed, built, and evaluated a working prototype that uses binary instrumentation to extract dynamic program information and passes it, via DMA, to a PCI mounted FPGA board for analysis.

The end goal of this research is three fold: 1) To demonstrate that there is potential benefit from specially designed program analysis devices, even if they can only be integrated with the system through existing peripheral channels. 2) To determine how "heavy-weight" a program analysis tool would need be before offloading has the potential for performance gains. 3) To identify and carefully quantify the bottlenecks and system-level issues in a real design so as to help identify places where microarchitectural support might be most judiciously applied. While we do not contend that this is always the best way to perform instrumentation and analysis, in some situations it may prove to be an enabling idea for those interested in building very analysis-heavy dynamic software analysis.

In this paper we describe the performance of the various components, the system-level problems that arise in

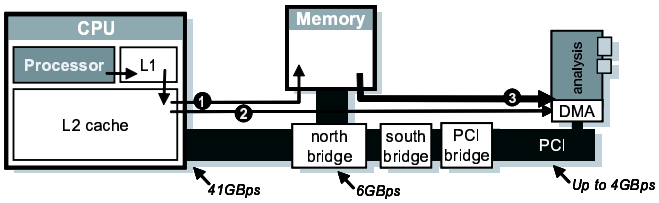

Figure 1. Figure shows important details of the online program analysis architecture. As a host CPU executes processes, profile is gathered via binary instrumentation. Gathered data is then batched in a buffer and handed over to the interconnect driver (here PCI) when it is full. The various stages of a DMA transfer are shown: 1. Data is written out to a buffer stored in memory. This buffer's pages are locked in physical memory to prevent them from being paged out. 2. Once the buffer is full, DMA setup writes are made to PCI registers to convey the size and address in memory of the buffer. 3. The PCI device then requests master control of PCI bus, and reads data from buffer. The figure also shows available points of attachment for an offchip analyzer and the available bandwidth at each point. A 10X increase in bandwidth can be seen as we move from a PCI(X) device to being attached directly to the CPU.

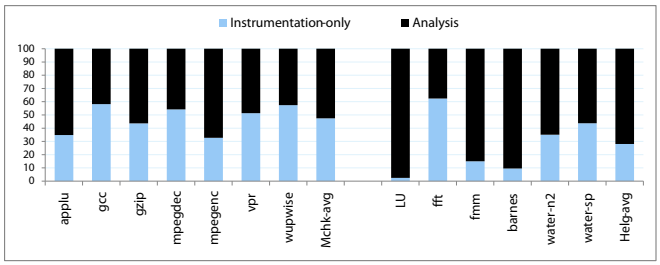

Figure 2. The ratio of analysis to instrumentation time in two software-only dynamic analysis tools, a memory error detector (Memcheck) and a race detector (Helgrind). The y-axis is percent time on a real executing system for the full tool (100%) versus a "stub" tool that excludes tool-specific analysis. Even the relatively simple Memcheck analysis takes a large portion of the execution time (over 50% on average), while Helgrind performs over 70% analysis on average.

getting the software elements to integrate seamlessly, a set of instrumentation optimizations specifically suited to this new profiling methodology, the problems with such an approach, and evaluate the architectural bottlenecks that prevent further scaling and provide opportunity for future research. Even with PCI technology developed 10 years ago, our prototype has the potential to speed up memory monitoring by 29% and a race condition detector, Helgrind, by 440% over the state-of-the-art, with room to grow using a better interconnect. However, before we can explore the potential of such an approach, we need to first understand the trends in peripheral bandwidth that makes this research possible.

II. Program Analysis on a Peripheral

Peripheral bandwidth and its application to program analysis: An important aspect of our approach is that it leverages existing market trends towards including high-speed board level interconnect. Figure 1 shows in detail the tremendous improvement in offchip bandwidth. A PCI-X bus (referred to simply as PCI throughout the rest of this paper) can support a theoretical peak bandwidth in the range of 133MBps to 4GBps, while a 32-channel PCI-e link (which is a point-to-point interface) can provide 6.4GBps peak bandwidth. Currently, peripheral devices can also be inserted into a HyperTranport Bus that acts as the Front Side Bus for the processor. Maximum throughputs currently for these 16b interconnects are 6.4GBps, and the processor-peripheral communication shares bandwidth with the processor-memory communication. At the highest end, FPGAs can be connected to an Opteron processor's open socket over a HyperTransport bus, at effective peak bandwidths of almost 12.8GBps for 16b transfers [10]. The HyperTransport-3.1 protocol released in 2008, increases the clock frequency from 1GHz to 3.2GHz and improves the 16b bandwidth to 25.6GBps. Moreover, by supporting an increased bus width of 32b, such a connection can attain a peak bandwidth of *51.2 GBps*.

The Overhead of Analysis vs Instrumentation: Even simple analysis, such as checking valid memory accesses, requires significant analysis computation. To demonstrate this point consider Memcheck [30], a memory monitoring tool implemented in the Valgrind binary instrumentation framework [19]. Memcheck [30] uses dataflow tracking to observe the memory usage of the monitored application and detects memory leaks, use of unallocated, undefined or freed memory regions and bad frees of heap blocks. It has been used in very large software projects such as the OpenOffice suite, Mozilla, KDE, GNOME, Perl, MySQL, Samba, Unreal Tournament and a host of other programs ranging in size up to 25 million lines of code [30]. We quantify the execution time break down on a real machine by executing Memcheck with and without the analysis functions inserted into monitored code and measuring the full and instrumentation only times.

Experiments indicate that a surprising amount of the slowdown comes from the actual *analysis overhead* rather than the *instrumentation overhead*. Specifically, Figure 2 shows that the computation required to check valid memory accesses accounts for more than 50% of the slowdown incurred by the Memcheck tool, while the rest is due to overhead of the Valgrind execution environment itself.

Race detection tools exhibit an even greater ratio of analysis. For example, Helgrind, a multi-threaded data race detector included in the Valgrind suite, implements the Lockset algorithm to expose errors in locking behavior of programs [19]. When used on Splash benchmarks [38] running only 4 threads, Helgrind slows down by almost 300X compared to native, with analysis accounting for 87.5% to over 99% of the overhead (for FMM and LU respectively) and instrumentation-only cost making up the rest. Similarly, Intel's ThreadChecker [25] has been shown

to have an average slowdown of 231X with 90% of it due to analysis. Clearly, the key to accelerating such tools is to target the analysis overhead.

Why an offchip accelerator: One natural question is, why not offload analysis to an alternate core available on a Chip Multi-Processor (CMP)? For example, Speck [21] is a software-only technique that transfers dynamic data to alternate cores in a CMP and uses a deterministic replay system to perform dynamic analysis in the background. For light-weight analyses like scanning the address space for sensitive data or analyzing system calls for anomalous behavior, this approach offers up to 7.5X speedup with 8 cores. However, we want to explore the possibility of eventually using hardware acceleration to run very heavy-weight analysis with more acceptable overheads and are less concerned with making very light-weight analysis run with 0% overhead. For example, Speck reports that the sequential nature of dataflow tracking severely limits the speedups to only 2X with 8 processors. While Speck improves the performance of Taintcheck with only 1 tag bit per word of memory, there is already a demand for tools that perform even more powerful analyses. Tools that do not simply detect errors but reveal their exact origin [29], the precise conditions that led to an exploit [37], or even help understand complex enterprise software [16]. These tools require *32-bit tags per byte* and incur analysis overheads from 100 to 300X.

If a specially designed device was developed to perform program analysis orders of magnitude faster than a general purpose processor, could we integrate it into a real system today with no changes to the processor? Where would the bottlenecks be, and how "heavy" does a tool need to be for off-chip acceleration to be a win? These are the questions that we attempt to answer in this paper. Later in the paper (Figure 5) we show that even with older technology, through our optimizations, less that 10% of the execution time on average is actually spent transferring the data off-chip. All of the other overheads of extracting the data should be the same regardless of whether the analysis is performed off-chip or on-chip, and quantifying them on a real system can point us to where micro-architectural support would be most useful. While in this paper we do not directly address the question of how that off-chip analysis hardware should be designed, there is no shortage of *on-chip* hardware-accelerators for analysis proposed in the literature to draw upon. Just as an example, Range Caches [34] can store a highly compressed representation of large dataflow tags, even the 32-bit tags per byte mentioned above.

III. Building an Analysis Peripheral

Instrumentation: We chose to use Memcheck and Helgrind as the dynamic binary analysis tools in our study. These are very popular tools included in the Valgrind instrumentation framework. We note that Pin [14] would have been an equivalent alternative to Valgrind, but Pin is closed source and did not work with the threading libraries used by WinDriver PCI driver. However, similar Pintools have been shown to have comparable overheads [5], and the choice between Pin and Valgrind does not affect our results.

Analysis logic on the FPGA: For our prototype analysis engine, we do not implement the complete analysis logic for both tools. Instead, we implement a hardware structure that forms the core of numerous dataflow tracking tools such as Memcheck[30], Taintcheck[20], Origin Tracking[29], Raksha[9] and many others. Since HARD [40] showed that hashing the lockset for an address into a 16-bit tag can enable efficient lockset processing in hardware, Range Cache could be used to store locksets too. Given the generic requirement of operating on large tags, we implemented Range Cache [34]. The additional logic needed to complete specific tools would be a logical OR of the tags for Memcheck and logical bit-operations for Helgrind that are easy to implement on FPGAs.

Interconnect: For our off-chip design we chose the most widely available technology, PCI, because it is well supported, the cost of development is low, and because it is sufficient to meet our objective of examining the feasibility and bottlenecks in peripheral-based profilers.

PCI has several different transfer modes, such as single transfers, batch transfers, and DMA. A single PCI write involves setting up a PCI transaction (read/write), and transfers one data word per transaction. This mode incurs the overhead of setting up the transaction for every word that is transferred, and thus has very limited throughput. The second option is to batch up the profile data in a buffer stored in the memory hierarchy, and perform a bulk transfer when it is full. In a bulk transfer, the PCI transaction is setup just once, and data is transferred on each subsequent clock cycle. This mode incurs several overheads, including allocating a buffer in Valgrind's segment of virtual memory, profile data writes to the memory buffer competing for space in the cache with useful program data, the time taken to write to and read back from the memory buffer and finally, performing a bulk write transfer over the PCI bus. The major gain this mode achieves is by avoiding the transaction setup time for each word that has to be transferred.

Figure 3 shows the performance of the above PCI-profilers compared to a software-only scheme. These result are, as all the results in this paper are, the actual timing data of a real implementation. Each data point is an average of 5 different runs. The software-only scheme is labeled *SW* in the figure and is on an average 90X slower than native program execution. The Y-axis represents the execution time of a PCI profiler as normalized to *SW* execution time for a variety of applications. As expected and is evident from Figure 3, single PCI writes (*PCI-Simple*) and bulk PCI writes (*PCI-Bulk*) suffer very high slowdowns (17X and 13X respectively) over the base software (*SW*) implementation.

The problem is that, even in a bulk transfer mode, the processor has to perform the tasks of writing out profile data per event to the memory buffer, read the buffer back when it is full, set up the PCI transaction and transfer the batched data over PCI. Hence, an immediate improvement can be made by freeing up the processor from two of these tasks: reading the data back from the memory and writing it out to the peripheral – using DMA.

In the simple DMA case, at each event (for Memcheck that is a load or a store) the processor makes a function call that writes data into the memory buffer. When the buffer is full, the PCI driver signals the DMA Controller on the PCI device to read the data from system memory. The PCI device then requests control of the PCI bus and reads the buffer from the main memory. This optimization makes a dramatic difference, reducing the runtime by almost a factor of 7 even if it uses this simple one-function call per event scheme (labeled *DMA-Func* in the graph). By inlining the function call and applying another optimization (discussed in Section IV), the 2X slowdown of *DMA-func* relative to *SW-only* reduces to almost 1.5X (labeled *DMA-Inline* in Figure 3). However, this is still slower than the software only approach.

The end point to take away here is that, while there is a potential for off-chip profilers to significantly speed program analysis, the most straight-forward approaches cannot match the performance of a software-only approach. To get to the heart of this slowdown, we have to more fully understand the way the instrumentation and the peripheral interact at the instruction level so that we can construct our system to avoid important bottlenecks.

IV. System-level Issues with Hardgrind

To explore the potential of analysis offloading, we have an implementation that uses a modified version of Valgrind and a PCI-mounted FPGA card (which we call Hardgrind). Our prototype is built using Valgrind-3.1.1 for Memcheck and Valgrind-2.4.1 for Helgrind on Linux. The instrumented binaries communicate profile data to the analyzer hardware implemented on a Stratix EP1S25F1020C5 FPGA plugged into a 32b/33MHz PCI slot. The analysis hardware has two components, Altera PCI Controller MegaCore function along with DMA logic forms the front-end of the device, while the Range Cache logic forms the analysis backend. A WinDriver PCI driver talks to the FPGA front-end from the software side.

The Range Cache synthesis results for the Stratix device indicate that it requires 1092 ALUTs (FPGA logic blocks) which is less than 10% of the total ALUTs on the FPGA, and has a maximum operating frequency of 160 MHz. Given that it requires two cycles to process a range request, the average throughput is 80M requests per second. If the memory accesses were being emitted in real time, this throughput is obviously insufficient. However, the slowdown imposed by simply offloading the analysis using DBI reduces the throughput requirement and makes it just enough.

The first key result is that even a slow interconnect and a slow analysis device provide immediate, substantial speedups. The second fallout is that if and when hardware support for offloading the data [6], [2] becomes available, extant higher-throughput interconnects and possibly novel analysis engines (ASIC, stream processor, or even general purpose cores) could use our techniques and obtain much higher speedups than we show in this work.

The key to implementing our prototype with low overhead is carefully optimizing the low level aspects of the instrumentation, managing the virtual address space, and exploiting the full potential of the DMA hardware interface. In this Section we discuss how the analysis peripheral can be mapped in the virtual address space of the program-under-test and how DMA interacts with this approach.

A. Mapping the Analysis Peripheral into Virtual Memory

The most fundamental job of our instrumentation scheme is to write the data out to the analysis device. The good news is if we can handle this functionality efficiently we can easily build support for many different types of analysis. The bad news is that we have to do a lot of these writes, so they have to be *very* efficient. A simplistic choice would be invoke `ioctls` for every write, but it is much better to map the PCI registers on the analysis device *into the virtual memory address space of the instrumented program*. The advantage of this is that we can then directly access the hardware through simple writes inserted (via instrumentation) into the program.

A client program is invoked in Valgrind by giving it as an argument to Valgrind's loader. The operating system invokes this loader, whose first task is to load Valgrind's initialization code into a high virtual address. After initializing the Valgrind core, it loads the specific Valgrind tool and its required libraries, and finally loads the client executable into the virtual address space. Virtual memory is assigned to each component as separate segments separated by "red zones" which are regions of memory purposely left empty so that overwrites can be detected.

In order to map the PCI device registers onto the same address space as the instrumented binary, the Valgrind core needs to reserve a segment of virtual addresses for communicating with the device. We modified Valgrind to lower the $clstack\_top$ value suggested to the client by 256MB, and the device driver is allocated segments from this reclaimed memory. The final allocated memory map can be seen in Figure 4, with space for the PCI registers, the PCI driver running in the kernel, buffer space allocated directly within Valgrind, and the instrumentation inserted into the program address space. Once this is done, the client executable is set to be translated and executed.

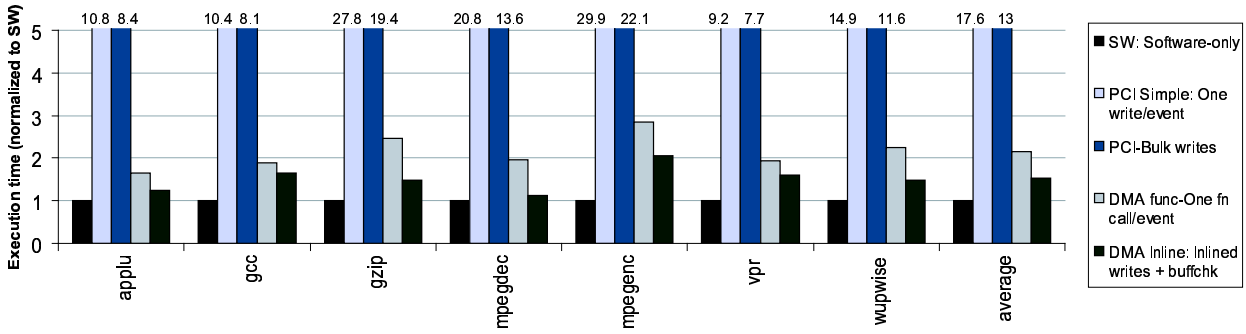

Figure 3. The figure compares performance of some preliminary PCI-versions of Memcheck to the base performance of software-only Memcheck. The PCI versions range from a simple one-PCI-write-per-event to an simple DMA-based Memcheck with all analysis code inlined as Valgrind micro-instructions. While these results indicate that with a simple DMA approach we can get close to the software-only results, a more clever scheme is required to beat that performance

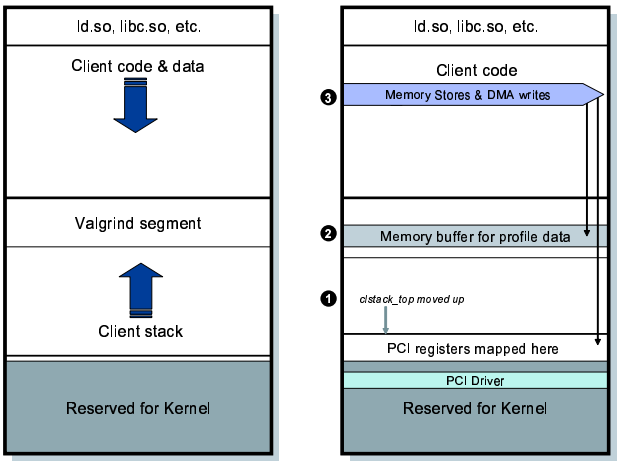

Figure 4. Virtual Address layout in Hardgrind: Figure shows the layout of the virtual address space shared by Valgrind, the PCI driver and the client program in addition to the kernel. (a)By default, the virtual address space is shared amongst the kernel, Valgrind itself and the client code. (b) The following changes are made to the default scheme: 1. The client's stack-top pointer is moved up to reclaim virtual memory where PCI registers can be mapped in. 2. A buffer to store profile data is allocated in Valgrind's segment of the virtual memory. 3. Code is inserted into the client's code segment to perform profile data stores to memory and DMA setup writes to memory-mapped PCI registers. Writes to the PCI registers are finally handled by a kernel driver module mapped into the kernel space.

One final note is that Valgrind prevents a monitored application from accidentally using a portion of the address space occupied by Valgrind data, by replacing malloc with a specialized version that avoids Valgrind addresses. It initializes its records with details about pre-existing memory-mapped segments by reading the contents of /proc/self/maps when it starts up. Valgrind can be made explicitly aware of the peripheral registers being memory-mapped into its address space by performing the PCI device initialization in the Valgrind core before it reads /proc/self/maps. Either the device has to be registered with Valgrind in this manner (so that its mallocs can avoid the peripheral's memory), or the PCI connection can be torn down and re-established in a new portion of the address-space if required. The address is not hard-wired into the PCI device, rather it is negotiated at initialization.

B. *Making efficient use of DMA*

As discussed in Section III, there are several different ways to access a PCI peripheral, and in the case of memory mapped devices, they all come down to writes. At first one might be tempted to simply perform write *directly* to the device. On the instrumentation side of things, this would be quite easy to implement: simply insert a store with the address of the device at every single instrumentation point. The problem is that, each of those writes is recognized by the system as an I/O access, the machine stalls, handles the event (writing it all the way out to the PCI bridge), waits for it to complete, and then resumes execution. What looks just like a simple store in fact incurs a huge performance penalty. The *PCI-Simple* approach from Figure 3 is actually implemented in this way.

Instead we need to make effective use of the memory hierarchy, buffering data in the cache until it is ready to be written. This involves utilizing the DMA capabilities of a PCI device, as shown in Figure 1. The processor just writes data into the memory buffer (which results in stores that slowly percolate out to main memory), and when the buffer is full, the PCI driver signals the DMA Controller on the PCI device to read the data from system memory (Two buffers are really required so that the profile data can be written into the second while the first one is being read out by the peripheral). The PCI device then requests control of the PCI bus and reads the buffer from the main memory (ensuring that main memory accesses by the processor take precedence). Specifically, the DMA procedure involves the following steps:

1) A buffer is allocated in *Valgrind's segment* of the virtual address space.
2) The PCI driver is used to 'lock' this buffer in physical memory. This is done so that the buffer does not get paged out to the disk when it is being read by the PCI

device. The Lock function also returns the physical page frame addresses for each virtual page of the buffer.

3) As the client executes, profile data is written into this locked buffer. When the buffer is full, the *instrumentation* module writes the physical address and size of the locked buffer to the PCI device.
4) The DMA logic on the device then requests mastery of the PCI bus. Given the size and the physical memory address of the profile data buffer, the device performs a Master Read of the main memory.

V. Optimizing Hardgrind Performance

Exploring the instrumentation framework and its interaction with the interconnect provides deeper insights towards realistic optimizations, opens new avenues for research in off-chip analysis engine, and aids in better architectural designs geared for heavy-weight program analysis. Towards this end, we demonstrate the utility of our PCI-based program analyzer with an actual tool (we use Memcheck to drive our design), carefully quantify the overheads, optimize the design and actually *beat* a software-only based approach, and then demonstrate that our results are applicable to other dynamic analysis tools. That is the goal of this section.

A. Quantifying the Overheads in the simple DMA scheme

When we first encountered the DMA results presented in Section III we were surprised. Freeing up the processor through *DMA-func* did not help improve Helgrind's performance beyond the software-only profiler that we use as our base. *DMA-func* provides two sources of overhead to be immediately optimized: one, the insertion of a function call after every event incurs the cost of saving and restoring the monitored program's registers' state; and two, checking if the buffer is full after every event involves executing a comparison followed by a conditional jump. We thus built an optimized version of *DMA-func* where the function call was replaced with the function code inlined directly into the monitored program's code. This inlined code stored profile data into a memory buffer and dynamically incremented the buffer's current pointer, without checking for the buffer being full. Buffer checking frequency was altered to be done once per basic block, instead of once per event, which substantially reduces the number of times its code is executed. Both these optimizations result in version we called *DMA Inline* which is illustrated in Figure 6. However, we know from the results in Figure 3 that the performance of this approach is still not sufficient. To more fully understand where the slowdown is coming from, we performed a series of experiments to clearly identify the sources of slowdown. At a high level, the overheads of *DMA-Inline* are due to its three components; instrumentation, data transfer, and analysis.

The instrumentation overhead stems from two sources. First, simply running a program (like gcc) as a Valgrind client incurs the overhead of its executable being translated into Valgrind's IR and back in a JIT manner. There is a default tool, *Nulgrind*, that does exactly that which allows us to calculate the overhead of that step. Second, a tool often provides its own version of certain functions and inserts listeners for various events detected by the Valgrind core. For example, Memcheck provides its own implementation of malloc, free and related functions, and registers handlers for system calls with the Valgrind core. Thus there is an added cost of just running the monitored program through a tool that has all its instrumentation setup in place but is not inserting any analysis code. For Memcheck, we term this second component *Memcheck_null*. As mentioned in Section II, we do not seek to reduce the cost of these two parts of instrumentation, but rather hope to reduce the cost of program *analysis*.

Since Hardgrind aims to supplant the cost of analysis with that of transferring data over to the offchip analyzer, the overheads come from: execution of the store to the profile buffer, the memory hierarchy interference caused by keeping the buffer in the user address space (i.e. cache pollution), the cost of incrementing the buffer pointer so that it does not overwrite the old data, checking if the buffer is full and ready to be written out to memory, and finally the cost of the actual DMA transfer itself. Figure 5 shows the contribution of each of the above components to *DMA-Inline*'s total execution time, for our set of benchmark programs. The method we used to calculate each breakdown is as follows:

Nulgrind and **Memcheck_null**: The two lowest segments in each stacked bar thus form the total instrumentation cost as measured through runs of *Nulgrind* and *Memcheck_null* tools that simply quantify the total instrumentation overhead.

Store instruction only – The third segment in the stack represents the overhead of inserting a store statement into the monitored program. To isolate just this overhead, we inserted store statements that access the *same* destination address at every event. Thus there should be minimal impact on the memory hierarchy (only one address is being accessed repeatedly) and there is no overhead in incrementing the pointer.

Increment instruction only – To determine the cost of *just* the increment operation, we need to exclude the effects of changing addresses on the system, which includes the cache pollution effects, the need to prevent buffer overflow, and the cost of flushing the buffer out over PCI. Our method of isolating each of these three overheads is by introducing the increment instructions into the monitored code, but with the increment value set to zero. This will quantify the time spent in executing the increment instructions and, with the buffer pointer staying at the same value, will not incur any memory hierarchy effects or buffer checking logic. This overhead is shown by the fourth segment from below in Figure 5.

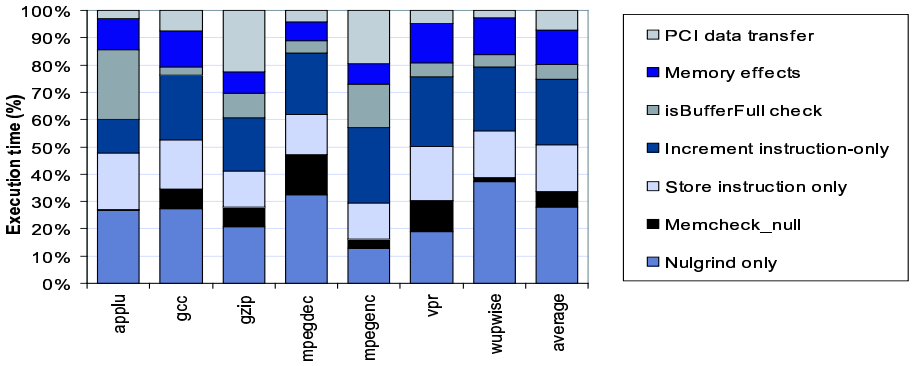

Figure 5. Breakdown of overheads in a *DMA-Inline* implementation: The cost of each component is shown as a percentage of total execution time. It is interesting to note that on average, the overheads of inserting a simple store instruction and incrementing the buffer address during run-time are individually greater than overheads due to memory pollution or data transfers to peripheral.

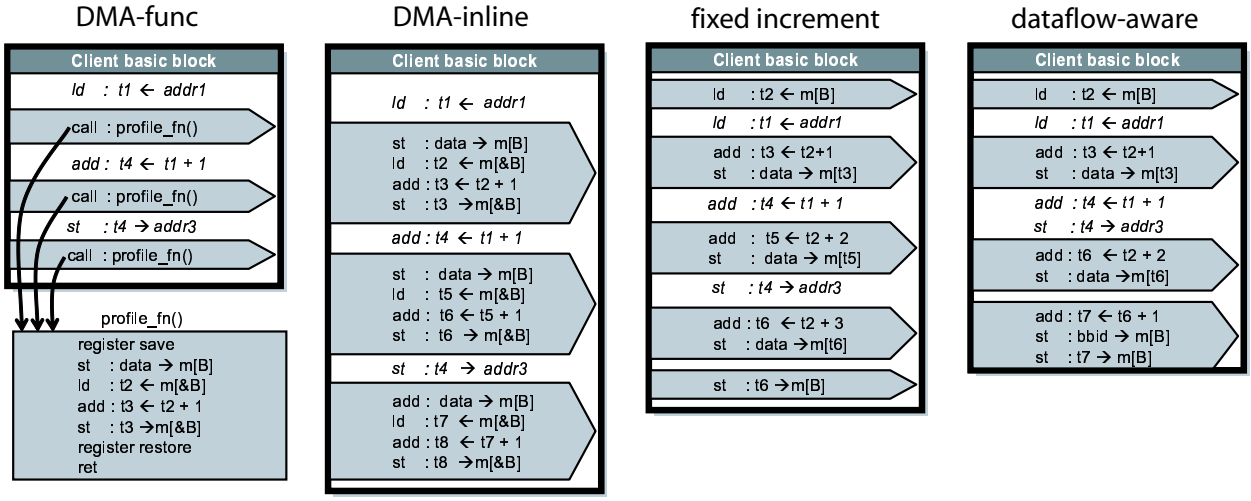

Figure 6. (L-R) The figure shows instrumented basic blocks for DMA implementations of Hardgrind, in increasing order of optimization. Client program's instructions are italicized and code inserted through binary instrumentation is included in shaded arrows. (a)*DMA-func* invokes a function call at every profiled event. This function stores profile data to a buffer address B, increments B, checks if the buffer is full, and if full, transfers it to the analyzer using DMA. (b)*DMA-Inline* inlines the above function code to be executed at each event to avoid register save and restore overhead. Additionally, it performs buffer overflow checks only one per basic block instead of once per event (not shown for simplicity).(c)*Fixed increment* saves on incrementing B during run-time by calculating store address offsets at instrumentation time for every store inserted into a basic block.(d)*Dataflow-aware* saves on inserting a store instruction at every event and instruments only loads, stores and basic block entries. It thus sends only the minimum run-time data required to reconstruct the execution of a basic block by the analyzer.

isBufferFull check – The next check is the instruction overhead of actually performing the check to see if the buffer is full and ready to be written out. We measure this by keeping all the stores still going to the same address, but the number of profile data writes per basic block is counted at instrumentation time and a check is performed at the end of the basic block as to whether the buffer is full or not (this is the buffer check optimization that was mentioned earlier – we check the buffer once per basic block rather than every instruction because we know exactly how many loads and stores are in a basic block at instrumentation time).

Memory effects – Now that we have tools that quantify the instruction execution overheads, we have increments and buffer checks in place, we can now add in the cache pollution effects to see what effect they have. All we have to do is change our tool to increment addresses by 1 word rather than by 0. The difference between the two should indicate the cache pollution effects on execution time.

PCI data transfer – The only difference between the tool to quantify the memory effects from an functional tool is that the data needs to be actually pulled from memory by the PCI device. Hence, as the last step, when the buffer is full, we insert stores to the memory mapped PCI registers to set up the DMA transfer. In our implementation, this requires two stores to communicate to the physical memory address and size of the profile data buffer to the PCI device. The execution time with the full DMA setup is represented by the whole bars in Figure 5, with the topmost segment indicating the overhead of performing the PCI data transfers.

There are several interesting things to note about these

results. The bottom two segments account for the cost of using Valgrind and the tool's specific requirements, and optimizing them is outside our project's focus. Of the remaining, the number of PCI DMA setup writes have already been minimized to send only the address and size of the buffer. In terms of the buffering, we observed that performance improvements from increasing the size of the buffer out to about 32KB, and we use this size for the rest of our experiments. Moreover, it is interesting to note that the *effect of cache pollution is on average less than the overheads of inserting either store instructions or the increment instructions*. While our initial idea was to propose some techniques to mitigate the impact on the memory hierarchy, after performing this analysis we realized that the overhead of the extra instructions themselves was the most important factor.

B. Optimized Design on Memcheck

To attack these overheads we have developed a set of optimization techniques that reduce the total number of instructions executed at each instrumentation point significantly.

Instrumentation-time Increment: The first target for optimization is the cost of increment instructions in the *DMA-inline* example above. In Figure 6 the cost of performing these increments at each and every instrumentation point becomes apparent (the additional instructions for buffer overflow checking (at the end of a basic block) and PCI DMA writes are not shown for simplicity). Instead, we note that because the code within a basic block will execute in that order, we can pre-compute the increments at instrumentation time. Rather than load the old value of the pointer, increment it, perform the store to that address, and then write back the updated pointer, this sequence of events can be replaced with a smaller set that makes use of the position of the instruction within the basic block.

The optimized code is labeled *Fixed Incr* in Figure 6. The example shows a monitored code sequence involving a load followed by an add and a store (shown in italics). If we assume Memcheck as the tool under consideration, all three instructions are 'events' that require analysis code to be inserted (to track both loads and stores but also to track the dataflow in the system).

The monitored program is instrumented in units of its basic blocks, and the number and location of buffer stores to be inserted in each block is known at instrumentation-time. In effect, the tool has an ordered list of buffer stores to be inserted, each of which addresses a successive address in the PCI buffer. In Figure 6 for example, Memcheck has to insert store instructions addressed to address (B+1) through (B+3), and increment B by 3 at the end of the basic block. This results in the run-time overhead being reduced to an add and a store per event, and an extra load and store per basic block (to get and save B). Figure 7 (second bar from left) shows the performance of *Instrumentation-time Incr* for various programs, and its reduced analysis cost is found to bring down its average execution time from 1.5X of *SW-only* for *DMA-Inline* to be almost on par with *SW-only*.

Dataflow-aware Profiling: Our final optimization to the Hardgrind architecture caters to tools that need to perform data-flow analysis and thus require extensive instrumentation for profile data gathering. Memcheck and Taintcheck are two important existing tools in this class, for they need to track definedness/validity/taintedness of data through both registers and memory locations. Such tools are most challenging for Hardgrind because of their high number of 'events' per time unit resulting in more code to be inserted and more data to be transferred.

As shown in Figure 5, even just having an extra store at every instruction results is a substantial overhead. Our optimization stems from the observation that, of all the dataflow that results from the execution of a basic block, it is only the loaded and stored addresses that cannot be determined at instrumentation time. The rest of the dataflow is a function of the structure of the basic block. Thus, for data flow analysis it is sufficient for a *Data-flow aware* profiler to send the address and a basic block identifier, as long as the analysis device can map the basic block identifier to a representation of the data flow effects that it would cause. Introducing this optimization into Hardgrind made a substantial impact to its performance, and resulted in an average speedup of *35%* over the previous optimization (*Instrumentation-time Incr*). Figure 7 shows the final results for Memcheck in software vs the various DMA schemes. With successive optimizations, Hardgrind improves from an almost *60%* slowdown in its *DMA-Inline* mode to being *on par* with *SW-only* in its *Instrumentation-time Incr* mode, and finally achieves a speedup of *29%* in its *Data-flow aware* mode. Moreover these trends of improving performance for successive optimizations and a final improvement over a software-only scheme are consistent for a varied set of benchmark programs.

C. Helgrind: Performance analysis

Thus far, we have demonstrated the effect of each optimization in Hardgrind on Memcheck. We observe modest performance improvements, and this shows that at around 50% ratio of analysis to instrumentation in the original tool, Memcheck forms the break-even point when the benefits of offloading the analysis will begin to show. Memcheck being one of the simplest dynamic information flow tracking tools today, this study indicates that tools such as Origin Tracking or 32-bit Taint tracking stand to gain much higher benefits (since they perform heavier analysis but use the same dynamic program data).

In this section we present a similar study of another important tool, Helgrind. Helgrind (a.k.a Valgrind-Lockset) is an implementation of the Eraser[27] data race-detection al-

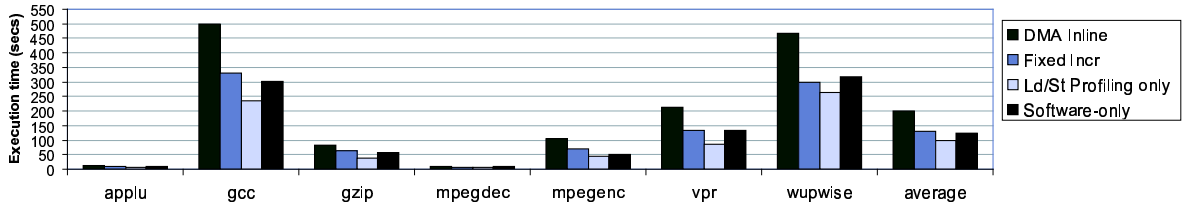

Figure 7. Performance of DMA implementations compared to software-only Memcheck: (L-R) Each cluster on the Y-axis represents execution times of *DMA-Inline*, *Fixed Increment*, *Dataflow-aware* and *SW-only* Memchecks. Successive optimizations from Figure 6 improve the DMA performance at each step, as seen in first three bars of each cluster. The final version of Hardgrind (second bar from right) has an average speedup of 29% over the rightmost bar of *SW-only* Memcheck.

gorithm for multi-threaded programs. Helgrind instruments only instructions that read or write to memory, and accesses tables of locks and locksets, for various threads, to ensure consistency in the locks held by these threads when a shared memory is accessed. Although the algorithm is simple, there is a significant amount of analysis computation involved when there is a lot of sharing (over 70% on average as shown in Figure 2).

We modified Helgrind to instrument a client basic block in the *Instrumentation-time Increment* fashion described in the previous section, and perform profile data stores at appropriate events (note that the dataflow-aware optimization does not apply to Helgrind because it already only instruments loads and stores). Every event requires the corresponding thread identifier and the memory location to be transferred. A performance comparison between *PCI Helgrind (Hardgrind)* and the *SW-only* version can be found in Figure 8. It demonstrates the potential of Hardgrind to accelerate the cases where there is heavyweight program analysis. We observed that applications such as LU, Barnes and FMM, which scaled very poorly in *SW-only* Helgrind for increasing processors and input sizes, consequently had the highest speedups. While a *SW-only* scheme suffers an average slowdown of 7X over Nulgrind, *Hardgrind* introduces only a 1.6X average slowdown, resulting in an overall speedup of 4.4X. Overall, we see that Hardgrind not only provides a platform to speedup existing analysis tools, but also opens up avenues for more complex analysis.

VI. Related Work

In addition to the work described throughout the paper, we can consider related work in one of two main camps, special purpose and general purpose.

Special Purpose: Memory errors are perhaps the most well studied set of special purpose problems. Statistical methods [4] and anomaly detection techniques for finding memory errors [3] are two examples. Relevant to our work, HeapMon [31] uses helper threads while MemTracker [36] provides support for attaching state bits to each virtual memory location to detect heap errors. MemorIES [17] proposes using an FPGA for accelerating memory system design by capturing and analyzing memory traffic, while iWatcher [39] provides mechanisms to watch for unsafe pointer dereferences and memory regions.

Hardware assistance can also be used dynamically track one-bit dataflow tags through the architecture [33], [7] to help identify the code that is most likely to be exploited by worms and other network based attacks. Furthermore, there is demand for even more complex tools that detect concurrency bugs such as data races [27] and atomicity violations [13], [11], [26], or record shared memory dependencies for later playback as in Strata [18]. HARD [40] proposes tightly integrated hardware acceleration of the lockset based race detection algorithm with 16 bit tags for cache lines and additional logic to update the tags. Using our techniques, custom accelerators could be built and fed through instrumentation rather than direct introspection.

General Purpose: Generalizing from these on-chip program profiling and analysis systems, several prior works have proposed methods to dynamically insert instructions into the execution stream [6] to alternate cores in CMPs [23], [32] to specialized profiling co-processors [41], reconfigurable monitors [28] and 3D introspection engines [15]. FlexiTaint [35] and Raksha [9] generalize uniprocessor hardware support for information flow tracking (Note that Raksha prototypes full systems on FPGAs, while we use FPGAs as coprocessors for existing systems). Patil and Fischer in [23] describe an approach close to ours where the computation of an instrumented program is divided into a main thread and a shadow thread where the shadow thread could potentially execute on an idle core in a multiprocessor architecture. The "shadow processing" approach can benefit from the instrumentation and transport mechanisms provided by our Hardgrind framework, and may not even require customization since a fast analysis engine on the FPGA has been shown to provide large performance benefits over general purpose cores for specific applications [8], [12]. Log-Based Architectures [2], [24] propose a similar decoupling in the context of CMPs, and uses hardware support to extract, compress and transport profile data to "lifeguard" processes running on idle cores of a multi-core processor. If a log-based architecture was implemented and sold as a commodity part, our technique would certainly benefit

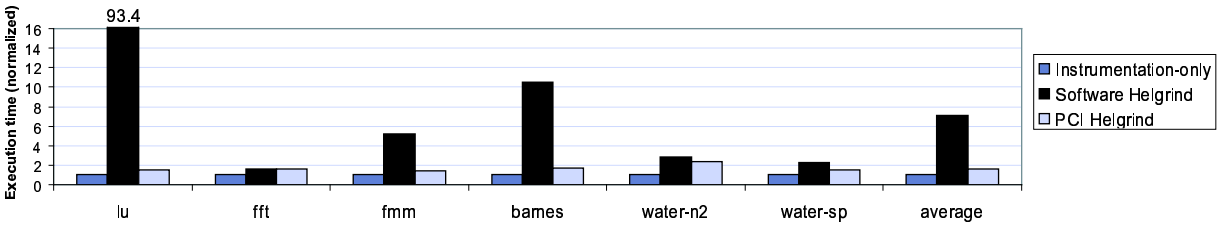

Figure 8. Performance comparison of Hardgrind and *SW-only* versions of Helgrind: The graph plots execution times of both versions normalized to Nulgrind's execution time on the Y-axis for different SPLASH-2 applications on the X-axis. As compared to Memcheck, Helgrind's greater ratio of analysis to instrumentation results in greater (4.4X on average) speedup for Hardgrind over *SW-only* Helgrind.

from the ability to extract information and compress it with less interaction from the core processor. With lower data extraction overheads, the new high-throughput interconnects could be exercised to their maximum throughput. The closest work to ours is Speck [21], that uses software techniques to offload analysis to alternate cores on CMPs. However, as mentioned in the motivation section, our approach is complementary in that we seek to accelerate very heavy-weight analyses that are hard to parallelize.

VII. Conclusion

Making software systems more bug-free, efficient, and secure has driven both academia and industry to develop powerful new dynamic analysis tools. As the burdens of system complexity, including parallelism, are passed along to the programmer, the need for a new class of tools has emerged. Such tools need not only to detect errors but must reveal information such as to the origin of those memory errors, the exact conditions under which a vulnerability was exploited, and uncover the thread interleavings that cause race conditions. Unfortunately running such tools on real-world software will be challenging since they run hundreds of times slower than native executions. With more than 90% of the time spent performing dynamic *analysis*, as opposed to the instrumentation to retrieve dynamic program data, the key to accelerating these tools lies in speeding up their analysis component. While previous studies have found that these heavy-weight analyses provide limited returns when parallelized using commodity multi-cores (2x on 8 cores) we consider a complementary approach and show that increases in I/O bandwidth can enable a new class of off-the-shelf hardware support for debugging: program analysis peripherals.

We demonstrate how, through careful optimization at the low level hardware/software interface, a dynamic instrumentation system can be used to shuttle data to an offchip analysis accelerator more efficiently than performing the analysis on the same processor core. Even for existing tools such as Memcheck and Helgrind (which each perform a moderate amount of run-time analysis) speedups of 29% and 440% respectively should be possible, using relatively slow hardware. This shows that a) there is a substantial improvement in performance to be had in the short term using only off-the-shelf components, b) future micro-architectural support should be directed towards flexible instrumentation such as DISE[6] or LBA[2](instead of mitigating impact of profile data on the cache hierarchy) while c) the increasing offchip bandwidth feeds specialized hardware that accelerate the analysis. In the end we hope that this paper motivates research into the development of programmable off-chip analysis accelerators, general purpose or otherwise.

Acknowledgments

The authors would like to thank Fred Chong and the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF Career Grant CCF-0448654.

References

[1] D. Bruening. Efficient, transparent, and comprehensive runtime code manipulation. In *PhD Thesis, M.I.T.*, Sept 2004.

[2] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 377–388, Washington, DC, USA, 2008. IEEE Computer Society.

[3] T. Chilimbi and V. Ganapathy. Heapmd: Identifying heap-based bugs using anomaly detection. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[4] T. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[5] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, 2007.

[6] M. L. Corliss, E. C. Lewis, and A. Roth. Low-overhead debugging via flexible dynamic instrumentation via dise. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 303–314, February 2005.

[7] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[8] Cray goes FPGA. http://www.fpgajournal.com/articles_2005/20050405_cray.htm.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *34th International Symposium on Computer Architecture (ISCA)*, 2007.

[10] DRC Reconfigurable Processor Unit Datasheet , 2006. http://www.drccomputer.com/pdfs/DRC_RPU100_datasheet.pdf.

[11] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[12] FPGA Acceleration in HPC: A Case Study in Financial Analytics. http://www.xtremedatainc.com/pdf/FPGA_Acceleration_in_HPC.pdf.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowner, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI'05*, Chicago IL, June 2005.

[15] S. Mysore, B. Agrawal, N. Srivastava, S.-C. Lin, K. Banerjee, and T. Sherwood. Introspective 3D chips. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 264–273, New York, NY, USA, 2006. ACM Press.

[16] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[17] A. Nanda, K.-K. Mak, K. Sugavanam, R. K. Sahoo, V. Soundararajan, and T. B. Smith. Memories: a programmable, real-time hardware emulation tool for multiprocessor server design. *SIGPLAN Not.*, 35(11):37–48, 2000.

[18] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[19] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, USA, June 2007.

[20] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[21] E. Nightingale, D. Peek, P. M. Chen, and J. Flynn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[22] NIST News Release. http://www.nist.gov/public_affairs/releases/n02-10.htm. 2002.

[23] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw. Pract. Exper.*, 27(1):87–110, 1997.

[24] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2008. ACM.

[25] P. Sack, B. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, 2006.

[26] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05*, pages 83–94, New York, NY, USA, 2005. ACM Press.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[28] M. Schulz, B. S. White, S. A. McKee, H.-H. S. Lee, and J. Jeitner. Owl: next generation system monitoring. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 116–124, New York, NY, USA, 2005. ACM Press.

[29] J. Seward. Origin tracking tool, valgrind release-3.4.0. Prerelease at svn co svn://svn.valgrind.org/valgrind/trunk.

[30] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, CA, USA, 2005.

[31] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50(2/3), 2006.

[32] W. Shi, H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, 2006.

[33] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. 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*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[34] M. Tiwari, B. Agrawal, S. Mysore, J. K. Valamehr, and T. Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the International Symposium on Microarchitecture (Micro)*, 2008.

[35] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[36] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[37] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, Italy, 1995.

[39] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2004.

[40] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[41] C. B. Zilles and G. S. Sohi. A programmable co-processor for profiling. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, 2001.