Exploring the Heterogeneous Design Space for both Performance and Reliability

Rafael Ubal, Dana Schaa, Perhaad Mistry, Xiang Gong, Yash Ukidave,
Zhongliang Chen, Gunar Schirner and David Kaeli
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA
{ubal,dschaa,pmistry,xgong,yukidave,zhonchen,schirner,kaeli}@ece.neu.edu

ABSTRACT

As we move into a new era of heterogeneous multi-core systems, our ability to tune the performance and understand the reliability of both hardware and software becomes more challenging. Given the multiplicity of different design trade-offs in hardware and software, and the rate of introduction of new architectures and hardware/-software features, it becomes difficult to properly model emerging heterogeneous platforms.

In this paper we present a new methodology to address these challenges in a flexible and extensible framework. We describe the design of a framework that supports a range of heterogeneous devices to be evaluated based on different performance/reliability criteria. We address heterogeneity both in hardware and software, providing a flexible framework that can be easily adapted and extended as new elements in the SoC stack continue to evolve. Our framework enables modeling at different levels of abstraction and interfaces to existing tools to compose hybrid modeling environments. We also consider the role of software, providing a flexible and modifiable compiler stack based on LLVM. We provide examples that highlight both the flexibility of this framework and demonstrate the utility of the tools.

1. INTRODUCTION

The days of single-core system architectures are clearly in the rear-view mirror. We are surrounded by multi-core architectures today, and we are designing for tomorrow's increasingly complex heterogeneous systems. This move is driven by a growing class of consumer applications that require both graphics and compute in smaller form factors, and in portable, power-efficient, systems.

Innovation in system architecture and embedded design has been driven by our ability to evaluate design trade-offs quantitatively. There have been a large number of efforts in the computer architecture community to develop accurate cycle-based simulators to study the value of a myriad of design features [7, 12, 13, 22, 32, 34]. Similar efforts have been undertaken in the embedded system community specifically with the emergence of System-Level Design Languages (SLDLs), such as SystemC [18], and frameworks have been developed for multicore and hardware software simulation including MParm [9], ReSP [8] and Daedalus [28].

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2596680.

In the past decade, we have moved away from simple homogeneous computing systems, where a single or a few cores on a few CPUs. Today we have computing fabrics that include CPUs, DSPs, GPUs and APUs. These systems range from high performance computing platforms, to full-featured smartphones. They include a range of inter-device relationships spanning master-slave to peer-to-peer, and everything in between. What is common in many of these environments is the need to run media/visualization along with compute, with software playing a larger role than ever before. In addition to these challenges, power and reliability are now treated as first-rate design concerns.

The rate of innovation for media-based devices is only limited by our imagination; however we need better tools to analyze compute, graphics, and an optimizing compiler technology. This need for sophisticated tools is a long standing requirement, as identified by Wolf in 1993 [36]. An architecture is needed for these tools that allow researchers and developers to efficiently evaluate the wide range of design decisions present in heterogeneous systems. We can no longer afford to develop stand-alone tools for particular architectures or bus interfaces. We also need to work seamlessly with existing tools, allowing designers to leverage past modeling efforts.

Multi2Sim currently provides a structural or cycle-based architectural simulation (or both) with full or partial support for seven different architectures, three CPUs and four GPUs, a range of memory configurations, network topologies, and runtime environments. These models have been developed over a six year period and are being used across a wide international community. Multi2Sim version 4.2 introduces new emulators for ARM and MIPS CPUs and the NVIDIA Fermi GPU, new network topologies and an optimized OpenCL runtime environment. This introduction builds upon a rich base that includes the original X86-based superscalar multi-core design models, AMD Evergreen (Radeon HD 5000 series) and Southern Islands GPU (Radeon HD 6000 series) architectures, and ongoing work on APUs, NVIDIA Kepler, and Analog Devices DSP architectures. The current tools support execution of C/C++ programs on the CPUs and OpenCL execution on the CPUs and GPUs. Ongoing work is targeting support for CUDA and the Heterogeneous Systems Architecture (HSA).

The common simulator architectural model of Multi2Sim has been leveraged by many different research groups, spanning high performance computing, server systems, tablet architectures and smartphone platforms. Examples include: GPU performance analysis [19, 19], CPU/memory power analysis [20, 24, 29], memory system analysis [27], GPU reliability [17, 31], graphics-compute inter-operability [35], and interconnect performance analysis [25]. Figure 1 shows the large design space we presently address with our infrastructure.

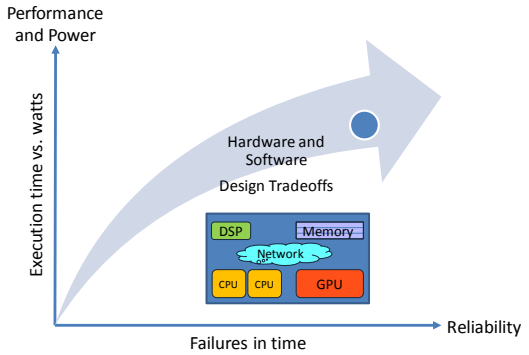

Figure 1: Multi2Sim allows users to explore design tradeoffs, comparing performance, power and reliability in a common framework.

We have developed an architecture that allows for the rapid development of modeling capabilities of new and evolving microarchitectures. Each architectural model uses a 4-stage development process (illustrated in Figure 2) to create tools that can act as standalone utilities or interact with each other: a disassembler, an emulator, a timing simulator, and a visualization tool. These are the key building blocks that allow us to study performance, power and reliablity on these platforms. The interfaces between these stages offer a unique level of generality that allows a programmer to intuitively provide plug-in software for new architectures.

In addition to our models delivered through this common simulator architecture, we utilize this same architected approach for developing Multi2C, our open-source compilation infrastructure. We are able to translate OpenCL and CUDA kernels to Clang (LLVM's intermediate representation). Multi2C support is available in the 4.2 release of Multi2Sim.

Multi2Sim supports the evaluation of a rich set of user benchmarks including SPEC2006, Mediabench, SPLASH-2, PARSEC2.1, and OpenCL SDK2.5 applications. Combined with a growing user/developer community, this open source project is impacting the work of a large number of researchers. Unmodified C, C++ and OpenCL applications run *out-of-the-box* on the Multi2Sim framework.

This paper presents some of the challenges involved in developing a modeling framework for next-generation heterogeneous systems, and provides some examples of our toolset's capabilities. The paper is organized as follows. Section 2 discusses prior work on developing heterogeneous modeling. Section 3 describes the Multi2Sim simulation architecture. Section 5 presents two case studies showing the flexibility of the infrastructure. Finally, Section 6 concludes the paper and discusses future directions.

2. RELATED WORK

Numerous simulation tools are publicly available. Regarding CPU simulation, SimpleScalar [12] has been one of the most popular tools to simulate superscalar processors. As architectures evolved, tools emerged to support multithreaded and multicore designs. SMTSim [34] and MSim [32] are extensions of SimpleScalar supporting multithreading. Simics [13] and GEMS [22] are widely used simulator of multi-core processors.

GPU simulation tools are as well available. Barra [14] is an ISA-level functional simulator (no timing model) targeting NVIDIA G80 GPUs and working at the ISA level. GPGPUSim [7] is a detailed GPU simulator including models for the memory hierarchy and interconnects. Ocelot [15] is an emulator of PTX intermediate code, with its own implementation of a CUDA runtime that dynamically compiles kernel binaries into CPU/GPU architectures.

Other tools are available to model critical hardware surrounding CPU/GPU cores. DRAMSim [30] is a public-domain DRAM system simulator, with ongoing extensions for cache and disk memory hierarchies. Topaz [4] and BigNetSim [6] are examples of interconnection network simulations, modeling the switch architectures with detail.

Compared to the aforementioned works, Multi2Sim is unique in that it models native ISA of multiple architectures, using unmodified, vendor-compliant binaries. It integrates architectural models for most major components of an APU, including CPU/GPU cores, memory hierarchy, and interconnects. It is integrated in an ecosystem of additional facilities to run simulations on clusters of computers, obtain packages of standard pre-compiled benchmarks, or participate in discussion on open forums. All this material is available for free on the project website [5].

3. MULTI2SIM FOR HETEROGENEOUS SYSTEMS

The Multi2Sim framework supports a number of CPU and GPU devices, both distributed (explicit communication) and shared memory environments. We first introduce the basic structure of the four key elements of any model in Multi2Sim and then highlight the modeling capabilities enabled by these four elements.

3.1 Multi2Sim Mechanics

Multi2Sim includes four key elements: *disassembler*, *emulator*, *timing simulator* and *visual tool*. They allow building a suite of tools serving a range of different purposes, including:

1. debugging heterogeneous applications utilizing program emulation,
2. identifying performance bottlenecks in a heterogeneous architecture,
3. characterizing the speedup available through GPU acceleration,
4. quantifying the reliability tradeoffs of both hardware and software optimizations, and
5. evaluating the role of the runtime and the compiler in heterogeneous system performance/power/reliability.

3.1.1 Disassembler

Interpreting programs at the Instruction Set Architecture (ISA)-level is essential for programming, debugging and accurate cycle-based simulation. Hence, a disassembler is necessary for adding new ISA model support. The dissembler works as follows:

1. consumes original program binaries in ELF format, generated by vendor-provided (or vendor-compliant) compiler, and
2. decodes each extracted instruction and provides the exact same output as the vendor disassembler.

While there are existing open-source tools to disassemble CPU binaries, most GPU architectures include these tools as part of their close-source drivers.

3.1.2 Emulator

The emulator runs a program (guest program) instruction by instruction, in an iterative process that reproduces what would be the program output on a real machine. The virtual state of the guest program is typically represented by its memory and registers. In each iteration of the main loop, the emulator fetches the next instruction from the program binary, reads the input operands from the virtual state, performs the encoded operation, and updates the

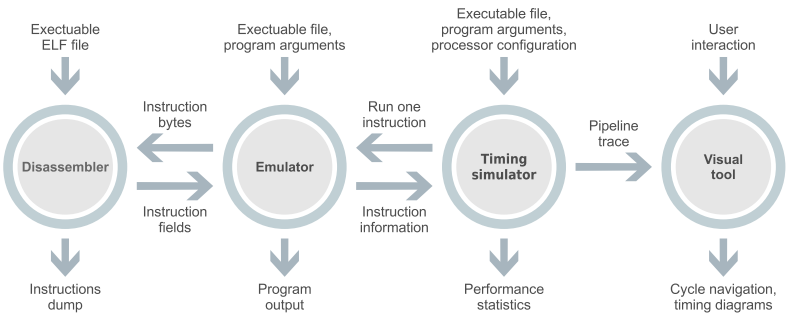

Figure 2: Four-stage design process for model development. Each tool can operate stand-alone, or serve as a library to tools on its right.

program state accordingly. Emulation offers a good amount of basic statistics and dynamic profiling information, such as instruction classification, or hot program regions.

3.1.3 Timing Simulator

The timing simulator models hardware components on a cycle-by-cycle basis, including instruction queues, functional units, register renaming tables, branch predictors, and more; all with configurable latency and geometry parameters. The configuration flexibility based on plain-text INI files makes our timing simulator suitable for architectural exploration, bottleneck detection, and fine-grain performance debugging. The timing simulator provides a very detailed statistics report, including utilization of hardware structures and timing results.

Timing verification involves two processes. First, short sequences of ISA instructions are assembled into a single long program that stresses one particular hardware component. After running the program both on real hardware and on the simulator, any small performance discrepancies can be precisely isolated. The second method consists in running real benchmarks covering most of the instruction set, again run on both real and virtual environments. Performance differences in this case are isolated and fed back to the first method iteratively, until all discrepancies have been addressed.

3.1.4 Visual Tool

An interactive visualization tool allows the user to navigate through the simulation using a scroll bar widget on a cycle-by-cycle basis. Its main window shows the set of CPU and GPU cores active during the simulation. The user can expand the level of detail and browse pipeline diagrams, instructions in flight, mapped threads, etc.. An additional panel explores the memory hierarchy, with details on cache sets, tags, blocks, or coherence states.

The visual tool is an invaluable resource to validate the timing simulator, especially when implicit communication occurs in the memory hierarchy on multi-core or GPU systems. We can visualize cache states and network messages as they progress through the system, dramatically improving visibility over plain-text logs.

3.2 Multi2C: Integrated Compiler Toolchain

Multi2C is an open-source compilation infrastructure integrated in Multi2Sim. It consumes OpenCL and CUDA kernel sources and generates executable binaries that can run on real GPUs. The compiler is architected modularly with i) front-ends that translate the kernel source code into the LLVM intermediate language [21], ii) back-ends that convert LLVM code into the target GPU ISA, and iii) assemblers that produce vendor-compliant binaries.

The front-end consumes an OpenCL/CUDA kernel source file and delivers LLVM 3.1 bitcode. Internally, our implementation leverages open-source tools *flex* [3] and *bison* [2] as lexical and syntax analyzers, respectively. An initial set of optimization passes can be applied using pre-defined LLVM library functions. Currently, the compiler front-end implements the complete OpenCL C grammar, while support for built-in functions is in progress. Multi2C has been tested with the AMD SDK 2.5 [1].

The back-end consumes an LLVM 3.1 binary module and produces plain-text assembly code, producing the final GPU ISA. LLVM is translated to ISA incrementally though a set of passes, starting with a one-to-many instruction equivalence. Three mandatory passes are applied to guarantee code correctness: *i*) generation of SIMD control-flow leveraging structural analysis, *ii*) a-posteriori translation of LLVM *phi* nodes, and *iii*) vector/scalar register mapping leveraging data-flow analysis.

The assembler reads a plain-text file and generates a vendor-compliant ELF binary. The main section on the input file contains the ISA code in the format established by the vendor, either through a public specification (AMD) or through its disassembler tools (NVIDIA). The remaining sections contain additional metadata that need to be encoded in the binary, such as program constants or kernel arguments. Currently we support the AMD Southern Islands backend. Ongoing development of back-ends and assemblers are currently in progress for the NVIDIA Fermi and Kepler target architectures.

4. HYBRID SIMULATION

Current modeling and design paradigms have to be expanded to allow designers to take advantage of GPUs in the embedded context. This includes performance estimation tools for emerging applications and extending design space exploration to include heterogeneous CPU/GPU processing. Most importantly however, with the increase in massive parallel compute power offered by the GPU, the pressure on the memory subsystem dramatically increases. Therefore, new methods are needed that explicitly model and analyze the GPU induced traffic.

To address the design challenges of GPU integration in the embedded space, we extend Multi2Sim to expose its detailed emulation capabilities to SystemC. Figure 3 presents one instance of Multi2Sim's hybrid modeling capabilities. In particular, we enable virtual platforms with a GPU model emulated in Multi2Sim, which is accessible through SystemC TLM 2.0. We pair an ARM Cortex A9 (emulated through OVP), which communicates through an AXI interconnect, with a GPU modeled in Multi2Sim. The OpenCL application on the CPU interacts with the GPU though the AXI bus. This allows for detailed analysis of the CPU/GPU traffic, as well as analyzing the GPU induced traffic.

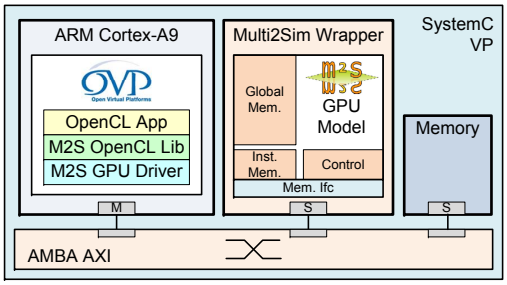

Figure 3: SystemC-based CPU / GPU virtual platform

In order to expose Multi2Sim to SystemC, we have created a SystemC wrapper which is connected through a standard SystemC TLM 2.0 to other components. Multi2Sim typically runs standalone, using its own discrete event simulation engine. For integration, we leveraged this simulation loop, making the GPU model callable to simulate a predefined quantum of Multi2Sim simulation steps. These are called inside the wrapper, and synchronize with SystemC time. To allow communication with the GPU model, a memory interface (*Mem Ifc.*) enables SystemC access to the memory emulated inside Multi2Sim. This allows us to access the global memory (for input and output data), instruction memory (to load the OpenCL kernel), as well as a set of control MMRs to synchronize and control GPU model execution.

The OpenCL application running on the ARM Cortex A9 controls and coordinates GPU execution. For this, new drivers are needed that provide CPU/GPU communication. This is realized using the Multi2Sim OpenCL run-time, which was originally developed for usage inside Multi2Sim. The OpenCL library uses a newly developed backend driver, the *M2S GPU Driver*.

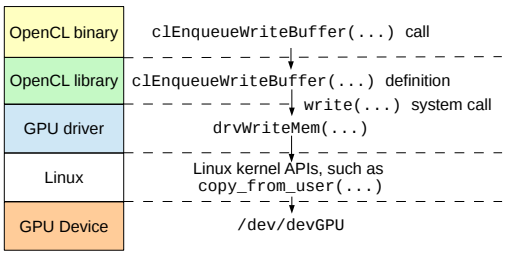

Figure 4: OpenCL Call Hierarchy

Figure 4 illustrates the call hierarchy in more detail. An OpenCL application consists of a host program and one or more device kernels. The host program can be written in a high-level programming language (e.g., C, Python) with the OpenCL extension, and linked with our OpenCL library. During its execution, the host program will dispatch OpenCL API functions (e.g., `clEnqueueWriteBuffer`). The OpenCL call passes control to its definition in the OpenCL library. To write data to the GPU device, `clEnqueueWriteBuffer` uses a write system call, defined as the function `drvWriteMem` in the GPU driver. Then several Linux kernel APIs (including `copy_from_user`) are called in `drvWriteMem` to copy data from the user-space OpenCL application to kernel-space device. The data is finally written to the GPU device `/dev/devGPU` transferring the data via the AXI interconnect.

With the Multi2Sim-SystemC integration, new research avenues are opened that combine the rich set of models available in Multi2Sim with the large set of available SystemC models. As such, it becomes feasible to investigate custom memory hierarchies, and interconnect schemes that are shared across a heterogeneous set of PEs (e.g. processors, GPUs, and custom hardware) considering actual bus traffic. Our Multi2Sim SystemC integration allows developers to explore the trade-off between CPU and GPU processing, as well investigate how to manage the traffic traffic demands due to heterogeneous processing. As such, Multi2Sim-SystemC is a valuable platform for virtual platform based design space exploration.

5. MULTI2SIM MODELS

5.1 Heterogeneous Performance Modeling

Multi2Sim has been used in hundreds of design space explorations that consider both CPUs and GPUs. The simulator can be used effectively for debugging OpenCL programs, identifying performance bottlenecks using the visual profiler, and can allow for the analysis of CPU/GPU interaction using shared memory in the latest release of the framework. We will focus on this last capability next in our first example of the versatility of the tools.

Ideally, memory coherence should be enabled for workloads that need it, and disabled for those that do not. In the age of heterogeneous many-cores (multi-cores with thousands of cores on-chip), the ability to enable/disable coherence could perhaps have significant impact on the performance and scalability. For GPUs, enabling coherence could potentially increase their utility to a larger variety of programs. For CPUs, disabling coherence when it is not required for correctness could have a dramatic impact on performance when executing highly data-parallel applications.

5.1.1 Memory Coherency Modeling

As CPU and GPU architectures cooperatively adapt to modern workloads, their memory systems —particularly the cache coherence protocols— play an increasingly important role. While a traditionally non-coherent memory hierarchy restricts the type of workloads executable on a GPU, a strictly coherent cache system on a CPU constrains performance of data-parallel applications, such as OpenCL or CUDA programs.

We have implemented a powerful enhancement to traditional coherence protocols (e.g., MESI, MOESI) in Multi2Sim. We have added the ability to model a non-coherent state, **N**, that allows for dynamic cache block transitions between coherent and non-coherent modes in a memory system.

To implement this in Multi2Sim, we modified the AMD Radeon 5870 memory system that presently implements a single, multi-banked L2, and incorporated multiple L2s that each serve a subset of the compute units. In the new design, each L2 is the size of a previous L2 bank, keeping the total L2 capacity constant. We also considered the design of the AMD Southern Island 7970 in this work, and have analyzed this protocol for additional target architectures given the flexibility of the Multi2Sim framework.

We found that removing unnecessary coherence operations for OpenCL programs can result in up to a 12% performance improvement on a CPU, and up to 31% on an APU. Providing optional coherence for GPUs also allows support for more diverse programs and relaxes memory system design constraints, providing up to 1.8X speedup on simulated benchmarks.

5.2 Heterogeneous Reliability

The impressive performance benefits provided by GPU computing have made them an attractive target for a range of important application domains. Biomedical image analysis, encryption/decryption and financial market analysis are just three examples of

the many general purpose applications now using GPUs. These three applications tend to be more sensitive to intermittent hardware faults, as compared to typical graphics workloads. These critical workloads demand the addition of fault tolerance mechanisms into future GPUs to ensure reliable computation.

Even though parity protection mechanisms are being incorporated by GPU vendors, and high-end GPUs are utilizing ECC in their memory systems, many of the GPU hardware structures remain unprotected. Recent efforts in this direction include software ECC for global memory [23], software thread-level and instruction-level replication for detection of faults in compute resources [16], and hardware-based checkers incorporated in the design at a fine granularity for low latency detection and recovery of faults in ALUs [33]. The common problem with these previous approaches is that they lack the required reliability characterization of the GPU hardware. It is becoming increasingly necessary to have more powerful protection techniques that cover other structures within a heterogeneous platform.

In our Multi2Sim modeling framework, we have introduced the ability to perform fault injection into the execution. Our strategy focuses on accessing the reliability of different microarchitectural structures. We have the ability to inject errors in to the runtime of any of our models. The fault injection mechanism is not specific to the microarchitecture, and can be applied to different GPU architectures. Given the overhead associated with fault-injection analysis, and the need to perform hundreds or thousands of injections to obtain statistical significance, we are presently developing the ability to directly compute the Architectural Vulnerability Factor (AVF) [11] of the targeted device. AVF analysis provides us with a measure the amount of vulnerable state per cycle. It captures the dynamics of the hardware and software to accurately identify if a bit flip in an hardware device will result in a program-visible error by the software.

5.2.1 Reliability modeling on an Evergreen GPU

To demonstrate the ability of our framework to study design tradeoffs when considering reliability, we present results from a recent study using statistical fault injection [17]. Our experimental evaluation is based on AMD's Evergreen family of GPUs. Our simulation framework allows us to modify parameters for the structures under study. We consider the impact on vulnerability as we change the total size, register width, local memory allocation granularity of different structures. In the experiments, we have used the AMD Radeon 5870. The benchmarks used in these experiments are from the AMD OpenCL SDK.

The fault-injection-based experiments were carried out using Multi2Sim. The simulator is fed a fault definition file which enumerates the faults to be injected during the simulation. Each line of the fault definition file contains the following information for a single fault: a) fault type, b) fault location, and c) fault injection time. Fault type specifies the structure where the fault should be injected; location is a bit-position within the structure; and time is the simulation cycle injected. Fault injection is simulated by a bit-flip in the desired position at the specified time. The faulty value remains intact, and potentially is propagated to other locations, until it is overwritten by the program. Multi2Sim supports multiple fault injections into multiple structures in a single simulation pass. For each structure, a total of 5000 single faults were injected.

In order to calculate AVF, we compute the number of fault injection experiments that resulted in a program failure, and divide by the total number of fault injected. All of the benchmarks have a self-check mechanism. This mechanism compares the output of the OpenCL kernel with a reference output calculated by the CPU.

These experiments (see Table 1) find that the AVF of the register file and local memory of the targeted architecture are 6% and 1%, respectively. These values are low, especially when compared to values reported on comparable CPU structures (AVF values are as high as 15% for the register file [26] and 25% for the cache memory [10]). While one might assume that the main memory on the GPU to be a large contributor to the overall system AVF, it was surprising to find that the AVF of the Active Mask Stack (AMS) (a much smaller memory structure) is 0.58% on average.

6. CONCLUSIONS

In this paper we have discussed some of the key challenges ahead for the design community when designing and simulating heterogeneous systems. We have presented the Multi2Sim simulation framework which allows users to build upon and leverage a 6-year effort that presently supports a wide range of users. We described some of the current work in the area of CPU and GPU architecture exploration. The two example studies considered the impact of changing the memory coherency protocol in an APU platform and modeling GPU reliability using fault injection.

In future work we plan to grow the Multi2Sim user/developer community, extending our work on compilation, SystemC integration, graphics interoperability, and programmable hardware (i.e., FPGAs). We also plan to focus our work on the emerging HSA standard that has been recently announced.

7. ACKNOWLEDGMENTS

The authors would like thank AMD, Analog Devices, NVIDIA, Samsung and Qualcomm for supporting this work. This work was also supported in part by NSF CISE grants CSR-1319501 and SHF-1017439.

8. REFERENCES

[1] AMD Accelerated Parallel Processing (APP) SDK. http://developer.amd.com/sdks/AMDAPPSDK.

[2] Lex: A Lexical Analyzer Generator. *Computing Science Technical Report No. 39, Bell Laboratories, Murray hill, New Jersey*, 1975.

[3] Yacc: Yet Another Compiler Compiler. *Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey*, 1975.

[4] P. Abad, P. Prieto, L. Menezo, A. Colaso, V. Puente, and J. Gregorio. TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers. In *Proc. of the 6th Int'l Symposium on Networks-on-Chip*, pages 99–106, May 2012.

[5] N. G. at Northeastern University. The Multi2Sim Simulation Framework: A CPU-GPU Model for Heterogeneous Computing. http://www.multi2sim.org/.

[6] P. P. L. at University of Illinois. Runtime Systems and Tools: BigNetSim—Parallel Interconnection Network Simulation. http://charm.cs.illinois.edu/research/bignetsim.

[7] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proc. of the Int'l Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009.

[8] G. Beltrame, L. Fossati, and D. Sciuto. Resp: A nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(12):1857–1869, 2009.

[9] L. Benini, D. Bertozzi, A. Bogoliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing*, 41(2):169–184, 2005.

[10] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan. Computing architectural

Table 1: Architectural vulnerability factor (AVF) and occupancy of register files (REG), active mask stacks (AMS), and local memories (MEM). *AVF-util* considers only the utilized portion of the corresponding hardware structure.

	REG(%)		AMS(%)	MEM(%)			MAX occupancy (%)		
Benchmark	AVF	AVF-util	AVF	AVF-util	AVF	AVF-util	REG	MEM	AMS
BitonicSort	0.04	25.00	0.00	0.00	N/A	N/A	0	0	3
DwtHaar1D	1.13	10.17	0.00	0.00	0.50	4.17	50	50	9
RecursiveGaussian	2.08	5.81	0.36	16.98	0.00	0.00	78	25	9
ScanLargeArrays	3.91	14.50	0.02	33.33	0.48	4.23	63	25	9
MatrixMultiplication	20.30	32.59	0.10	71.43	3.75	3.84	63	100	13
SobelFilter	19.36	22.50	2.86	24.78	N/A	N/A	75	0	9
DCT	3.68	9.01	0.72	53.73	0.10	1.74	44	6	6
URNG	0.18	0.88	0.00	0.00	N/A	N/A	19	0	3
Average	6.34	15.06	0.58	40.05	0.97	2.80	49	23	8

vulnerability factors for address-based structures. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 532 – 543, june 2005.

[11] A. Biswas, P. Racunas, J. Emer, and S. Mukherjee. Computing accurate avfs using ace analysis on performance models: A rebuttal. *Computer Architecture Letters*, 7(1):21 –24, jan. 2008.

[12] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, 1997.

[13] M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[14] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *Proc. of the 18th Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug. 2010.

[15] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proc. of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, Sept. 2010.

[16] M. Dimitrov, M. Mantor, and H. Zhou. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 94–104, 2009.

[17] N. Farazmand, R. Ubal, and D. Kaeli. Statistical fault injection-based avf analysis of a gpu architecture. In *IEEE Workshop on Silicon Errors in Logic*, 2012.

[18] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, 2002.

[19] B. Jang, P. Mistry, D. Scha, and D. Kaeli. Static memory access pattern analysis on a massively parallel gpu. In *Proceedings of SAAHPC*, 2010.

[20] C.-Y. Lai, G.-Y. Pan, H.-K. Kuo, and J.-Y. Jou. A read-write aware dram scheduling for power reduction in multi-core systems. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 604–609, Jan 2014.

[21] C. Lattner. LLVM and Clang: Advancing Compiler Technology. *Free and Open Source Developers European Meeting (keynote)*, Feb. 2011.

[22] M. R. Marty, B. Beckmann, L. Yen, A. R. Alameldeen, M. Xu, and K. Moore. GEMS: Multifacet's General Execution-Driven Multiprocessor Simulator. *Proc. of the 33rd Int'l Symposium on Computer Architecture*, pages 92–99, June 2006.

[23] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity gpus. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[24] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[25] R. Mohanty, A. Turuk, and B. Sahoo. Performance evaluation of multi-core processors with varied interconnect networks. In *Advanced Computing, Networking and Security (ADCONS), 2013 2nd International Conference on*, pages 7–11, Dec 2013.

[26] P. Montesinos, W. Liu, and J. Torrellas. Using register lifetime predictions to protect register files against soft errors. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 286 –296, june 2007.

[27] R. Natarajan and M. Chaudhuri. Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 1–10, Sept 2013.

[28] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: Toward composable multimedia mp-soc design. In *DAC*, pages 574–579, 2008.

[29] S. K. Rethinagiri, O. Palomar, R. Ben Atitallah, S. Niar, O. Unsal, and A. C. Kestelman. System-level power estimation tool for embedded processor based platforms. In *Proceedings of the 6th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '14, pages 5:1–5:8, New York, NY, USA, 2014. ACM.

[30] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle-Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, Jan. 2011.

[31] R. Shah, M. Choi, and B. Jang. Workload-Dependent Relative Fault Sensitivity and Error Contribution Factor of GPU On-Chip Memory Structures. In *Proc. of the Int'l Conference on Embedded Computer Systems*, pages 271–278, July 2013.

[32] J. Sharkey. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. *Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton*, 2005.

[33] J. W. Sheaffer, D. P. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 55–64, 2007.

[34] D. M. Tullsen. Simulation and Modeling of a Simultaneous Multithreading Processor. *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[35] Y. Ukidave, X. Gong, and D. R. Kaeli. Performance evaluation and optimization mechanisms for inter-operable graphics and computation on gpus. In *GPGPU@ASPLOS*, page 37, 2014.

[36] W. Wolf. Guest editor's introduction: Hardware-software codesign. *IEEE Design & Test of Computers*, 10(3):5–, 1993.