Scalable LLVM-Based Accelerator Modeling in gem5

Samuel Rogers, *Student Member, IEEE*,
Joshua Slycord, *Student Member, IEEE*,
Ronak Raheja, *Student Member, IEEE*,
and Hamed Tabkhi, *Member, IEEE*

Abstract—This article proposes a scalable integrated system architecture modeling for hardware accelerator based in gem5 simulation framework. The core of proposed modeling is a LLVM-based simulation engine for modeling any customized data-path with respect to inherent data/instruction-level parallelism (derived by algorithms) and available compute units (defined by the user). The simulation framework also offers a general-purpose communication interface that allows a scalable and flexible connection into the gem5 ecosystem. Python API of gem5, enabling modifications to the system hierarchy without the need to rebuild the underlying simulator. Our simulation framework currently supports full-system simulation (both bare-metal and a full Linux kernel) for ARM-based systems, with future plans to add support for RISC-V. The LLVM-based modeling and modular integration to gem5 allow long-term simulation expansion and sustainable design modeling for emerging applications with demands for acceleration.

Index Terms—Computer architecture simulation, hardware accelerators, heterogeneous systems

1 Introduction

Having hit the limitations of general purpose processing, engineering efforts have shifted toward application/domain-specific accelerators. To simplify and enhance design productivity, new design tools and simulation platforms are required to enable quick design space exploration of accelerators and entire system interactions (memory hierarchy, communication, and synchronization). Examples, such as cgem5 [1], gem5-Aladdin [2], and PARADE [3], support cycle-level hardware modeling within the gem5 [4] system simulation framework in order to model system-level interactions. cgem5 offers a generic SystemC wrapper, to integrate SystemC models of hardware accelerators to gem5 system architecture simulator [1]. However, the scope of system-level modeling is limited to PCI-based discrete hardware accelerators with dedicated private memory address space in order to simplify the communication between SystemC and gem5. More recently the main gem5 project has been updated with direct SystemC support through TLM translation[5]. This enables the exploration of a much larger variety of systems, but requires significant knowledge and investment in SystemC development.

The gem5-Aladdin [2] simulator integrates a runtime trace-based simulator, ALADDIN [6], into gem5 with support for both private scratchpad memories (SPM) and coherent caches. The trace-based modeling of accelerators, at the core of ALADDIN, imposes several limitations which narrow the scope of system-level modeling with multiple hardware accelerators. The most significant limitation is the implementation of private SPMs. In order to guarantee the data availability assumed by the trace-based simulation, these SPMs are not exposed to the gem5 memory system. Instead, they are managed by a DMA device fused with the accelerator model and controlled through the addition of DMA load/store operations to the accelerator execution graph. Since DMA transactions are managed by each accelerator, rather than a central controller, any intra-device communication and synchronization must instead pass through the accelerator cache ports. This makes common multi-device optimizations such as parallel producer-consumer structures impractical to implement.

In an orthogonal approach, the PARADE [3] simulator attempts to explore scheduling and communication in large, accelerator-rich systems. This approach trades accelerator modeling accuracy, for the opportunity to explore interactions between accelerators in a large heterogeneous system. PARADE exchanges datapath modeling for a function-level model instrumented with static timing, power and area values from HLS. While RTL modeling can provide early insights into device performance, RTL simulation does not account variation in the performance of irregular applications with data-dependent controls. Additionally, the non-deterministic memory latency associated with memory elements, such as cache, are incompatible with static timing models. Furthermore, system-level connectivity and task scheduling are abstracted from the user through a custom data flow language that constrains system integration to the base architecture model described in [7].

Overall, current tools impose several system-level design assumptions that limit the scope of design space exploration of ACC-rich SoCs. This article proposes a novel approach for scalable pre-RTL design space exploration of application-specific accelerators.[1] It offers custom datapath modeling by leveraging the LLVM toolchain and intermediate representation (IR). Fig. 1 illustrates our proposed LLVM-based simulation infrastructure. Unlike other popular simulation tools, the LLVM-based hardware model is a direct extension of gem5. It consists of the LLVM runtime engine and a general communications interface to provide the full system timing, interconnect modeling and memory hierarchy design of hardware accelerators. This integration also includes a synchronized timing mechanism built on top of the gem5 event scheduling system. It offers a cycle-accurate simulation model for hardware acceleration of any given application. Users have the ability to generate the entire system with multiple hardware accelerators and different system-level configuration using gem5-python features.

2 LLVM-Based Data-Path Simulation

The core of the proposed approach is a LLVM-based simulation of data-path for any given application. Fig. 2 presents the overall simulation architecture of our proposed LLVM-based application-specific simulator. Our LLVM-based accelerator simulation engine is built from scratch as a new "SimObject" within gem5. It offers a cycle-accurate hardware accelerator simulator tied into the gem5 architecture. It has two major component: (1) Static LLVM analysis and parser for generating accelerator model, and (2) dynamic runtime engine.

2.1 Accelerator Model Generation

Accelerator model generation starts with a description of the accelerated function in C or C++. This description must be compatible with HLS synthesis, with helper functions inlined and large arrays passed in as global pointers. This code is compiled to the LLVM IR via Clang. During instantiation of the accelerator within the simulation environment, a static Control and Data Flow Graph (CDFG), like the one shown for vector add in Fig. 3, will be generated from the LLVM

• *The authors are with the Department of Electrical and Computer Engineering, University of Noth Carolina, Charlotte, NC 28223. E-mail: {sroger48, jslycord, rraheja, htabkhiv}@uncc.edu.*

Manuscript received 23 Oct. 2018; revised 25 Dec. 2018; accepted 3 Jan. 2019. Date of publication 21 Jan. 2019; date of current version 7 Feb. 2019.
(Corresponding author: Samuel Rogers.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2019.2893932

1. The simulator is publicly available online at https://github.com/TeCSAR-UNCC/gem5-SALAM.

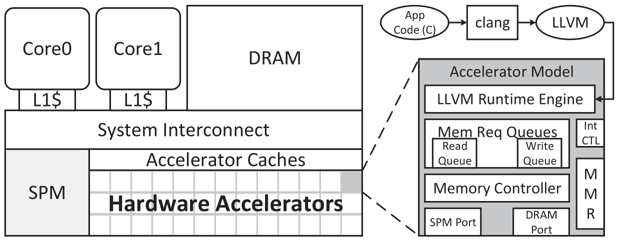

Fig. 1. gem5 system integration.

IR. Any CDFG optimizations such as CDFG simplification, vectorization, and loop unrolling should be performed by the compiler.

To take advantage of the data-flow behavior of hardware accelerators, our custom LLVM Parser, shown in Fig. 2, reads the LLVM IR file and performs the task of CDFG extraction and functional unit allocation. We consider each instruction, or node in our CDFG, as a functional unit in a data-path. Function unit reuse can be imposed as a configuration option of the runtime engine. Connectivity between these functional units is determined by their source and destination registers. The resulting datapath captures all statically defined spatial and temporal parallelism in the source LLVM file, including vectorization and loop unrolling.

2.2 Dynamic Runtime Engine

Dynamic control and parallelism within this static datapath are managed by the scheduler and reservations queues during simulation execution. The right side of Fig. 2 provides an overview of our runtime model. It consists of "Runtime Scheduler", "Reservation Queue", and "Event Queue".

Runtime Scheduler. In order to simulate the behavior of a pipelined datapath, a dynamic CDFG of the application is generated by a runtime scheduler. Runtime scheduling of our static CDFG is performed at the granularity of basic-block. During this phase, a dynamic version of each node is generated from its static counterpart and added to a runtime reservation queue. Dependencies are added to these nodes by performing a reverse search of our reservation and event queues, for the last instance of any runtime node that generates a dependency. These dependencies are represented by the solid line in Fig. 4. If a dependency is not found in our reservation or event queues, we assume that the register associated with that dependency is safe to read, and immediately fetch its value. Additionally, we impose the constraint of one function unit per unique instruction by drawing a dependency to the last instance of the same node, signified by the dashed lines in Fig. 4. This enables loop pipelining at runtime. The scheduler is also responsible for managing pipeline stalls associated with cache misses and high latency computation, only blocking pipeline lanes that rely on the high latency operation.

Reservation Queue. Upon transfer to the reservation queue (Fig. 2), we check each node for active parent dependencies. Normally, if a node does not have any active parents, we transfer it to the appropriate event queue for execution. We can modify this behavior by

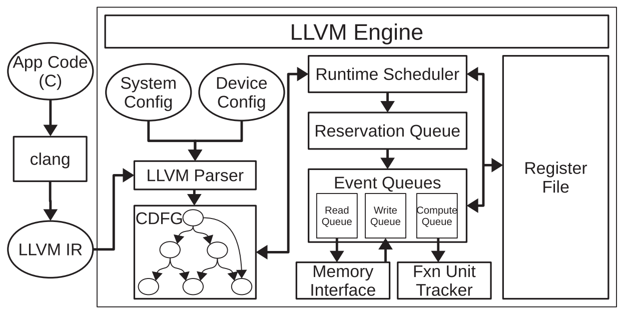

Fig. 2. LLVM-based data-path modeling.

```
0.  %i = PHI (0, i++)
1.  %1 = GEP %a, %i //addr a[i]
2.  %2 = LD %1 //load a[i]
3.  %3 = GEP %b, %i //addr b[i]
4.  %4 = LD %3 //load b[i]
5.  %5 = ADD %2, %4
6.  %6 = GEP %c, %i //addr c[i]
7.  ST %5, %6 //store c[i]
8.  %i++ = ADD %i, 1
9.  %exit = CMP %i++, N
10. BR Loop, Next
```

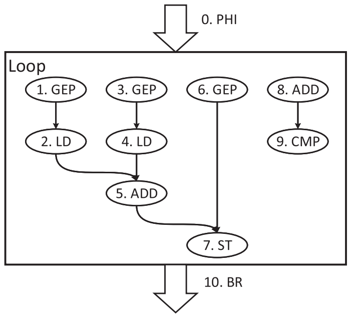

Fig. 3. Mapping VAdd to a datapath.

limiting the number of functional units in our data-path. Functional unit parametrization is exposed through our Python configuration. In this way, we can enforce the sharing of expensive functional units such as floating point multipliers, which is commonly done in hardware optimization. Lastly, we recognize that many LLVM instructions do not translate to actual hardware. Thus, we set their execution to be zero-cycle, such that the control flow of the algorithm is maintained without adding false hardware overheads.

Event Queue. The event queue is dynamically assigned compute nodes to process, and each node is subsequently placed into an appropriate sub-queue that is either a memory operation or a computational operation. Nodes within these queues are checked for at the start of each cycle for commit. Upon commit, any child nodes are signaled to clear the dependency and fetch the associated register. The event queue contains memory read and memory write sub-queues exist that interface directly into gem5 and track in-flight memory operations. This integration is described in the next section. The en-queued compute nodes that do not require memory access are passed into our simulators computation engine that performs the operation, stores the results back into the register file, and updates the current functional unit usage statistics.

3 GEM5 INTEGRATION

Key components in our model are scalability and the ability to explore large ACC-rich heterogeneous systems based on gem5 simulation semantic. In the following, we briefly introduce the communication interface, memory hierarchy, system-level simulation flexibility, and interface with application processor.

Communication Interface. Fig. 5 presents the communications interface. It provides a flexible communications infrastructure for the accelerator model through a programmable memory-mapped register (MMR), an interrupt line interfaced with the ARM generic interrupt controller (GIC) model, and a multi-ported memory interface capable of connecting to both private scratchpad memories (SPMs) and shared memory resources (cache, SPM, DRAM, etc). This interface was constructed using the gem5 Peripheral I/O Device (PioDevice) template in order to model the behavior of a memory-mapped device. The direct integration into gem5's system interconnect allows for advanced synchronization techniques between accelerator instances as well as other devices within

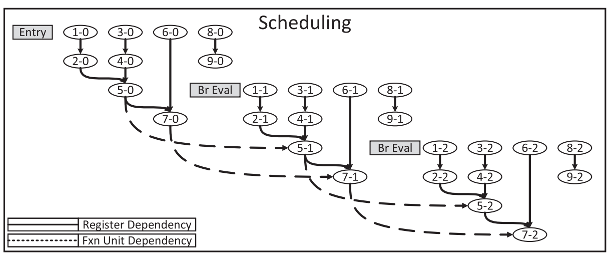

Fig. 4. Runtime scheduling.

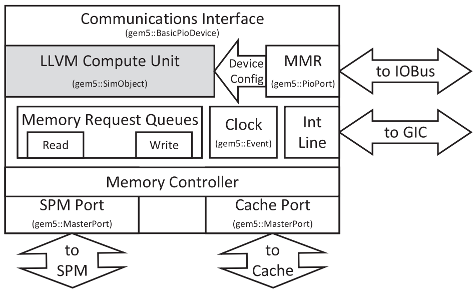

Fig. 5. gem5 accelerator model.

a simulation instance. This enables more advanced domain-level design space exploration that leverages multiple heterogeneous accelerators combined with other processing elements to perform a broader range of tasks.

Memory Hierarchy. Fig. 6 shows a system hierarchy currently available in our simulation environment. Our memory modeling enables rapid prototyping of heterogeneous application-specific memory hierarchy, as proposed in [8]. Memory modeling in this system is broken into three abstractions: private, local, and global. Each accelerator has its own private memory. Externally this memory looks like any other scratchpad. From the internal accelerator perspective, this memory can have the form, i.e., wide multi-port, streaming line buffer, etc., that is best suited to the accelerator's task. Each accelerator also has access to a local memory bus. This bus provides access to on-chip devices within the same cluster such as SPM and other processing elements. This enables inter-device communication without needing a host CPU. Lastly, all accelerators can be connected to a shared cache that provides coherent access to main memory and other clusters. When the device generates a memory request it checks which address range the request falls into and sends it out of the corresponding port. The system port, which connects to the cache or DRAM, handles any requests that fall outside of the private or local ranges.

Extensibility and System Configuration Flexibility. A key design goal in developing the gem5 integration was to embrace the modular inheritance-based design of gem5. In the same way that CPU simulation in gem5 is built upon modular device templates that provide a framework for future extensions, the components of our accelerator models are designed to be extended. Devices built upon our basic ComputeUnit, such as the LLVM runtime, can integrate seamlessly with our communications infrastructure (CommInterface). Similarly, extensions to the basic communications infrastructure can integrate seamlessly with elements like the LLVM runtime engine. The accelerator is integrated into the gem5 infrastructure via the Python configuration scripts. The gem5-python configuration is used for connecting the device within the simulated system, and basic configuration of the device (e.g., number of memory ports, operating frequency, MMR addresses, etc). Additionally, each device has config. parameters for local memory range (cluster memory address range) and private memory range (private SPM). The configuration of clusters, like the one shown in Fig. 6, is fully exposed in the Python API of gem5, enabling modifications to the system hierarchy without the need to rebuild the underlying simulator.

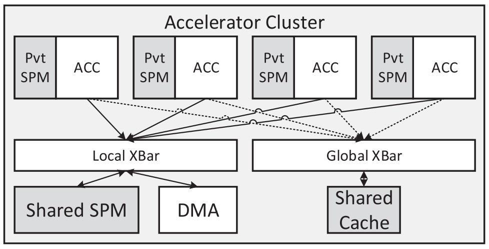

Fig. 6. Integration example.

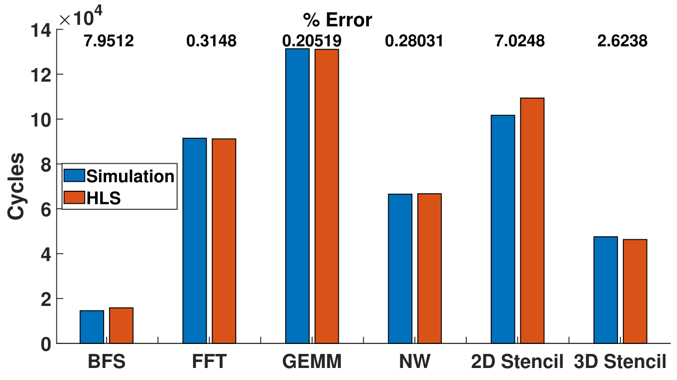

Fig. 7. Comparison of LLVM runtime model v. RTL simulation.

Interface with Application Processor. The scope of acceleration is at function granularity. The portion of code that a programmer wishes to accelerate is isolated from the primary host code, and compiled to LLVM separately. As mentioned earlier, the accelerator is integrated into the gem5 infrastructure via the Python configuration scripts. This integration involves defining a memory address range for the device's memory mapped registers as well as any scratchpad memories (SimpleMemory or our PrivateMemory in gem5) as part of the physical memory address space when they need to be managed by the host code running on CPU models. The original function call in the host code is replaced with an appropriate driver call for the accelerator device. The driver is responsible for configuring the hardware device's MMRs as well as invoking the DMAs to copy any data that isn't being accessed through caches, to the accelerator's SPM.

4 Early Results and Evaluation

We evaluated the timing accuracy of our model against RTL simulation for the MachSuite[9] hardware accelerator benchmarks. For RTL simulation the benchmark source files were synthesized using Vivado HLS v2017.4. These synthesized RTL models were then evaluated for timing using ModelSim from the Intel Quartus toolchain. For our simulation, we used the clang front-end compiler to generate LLVM for the same benchmarks. We applied the same set of optimizations as HLS to our files. Vivado HLS also uses clang as part of its C-to-RTL translation, so the resulting LLVM IR file should be roughly identical to what Vivado HLS uses when mapping our hardware. Since RTL simulation does not factor in off-chip to on-chip data transfer in its timing model, we evaluate the timing of our simulation with the assumption that all data is already in an on-chip scratchpad memory. Timing evaluation for our simulation and RTL simulation assumes a 10ns clock period.

Fig. 7 shows the result of our evaluation for six benchmarks from MachSuite: Breadth-First Search (BFS-Bulk), Fast-Fourier Transform (FFT-Strided), General Matrix Multiplication (GEMM), Needleman-Wunsch (NW), and 2D/3D Stencil computation. The highest timing error of 7.95 percent was associated with BFS. This application contains many runtime data-dependent control operations, accounting for 30 percent of executed instructions. 2D Stencil, with the second highest error of 7.02 percent was another control intensive application (18 percent of instructions), with deeply nested loops. In both cases HLS appears to have taken a more conservative approach toward pipeline synthesis in order to allow time for control signals to propagate. In contrast the difference between simulation and synthesis results for NW was quite small. NW is another heavily control-intense application (25 percent of instructions), however many of these control instructions can be evaluated in parallel, a

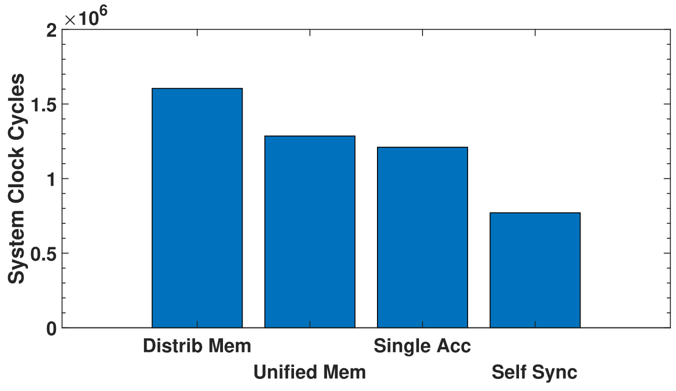

Fig. 8. Comparison of accelerator system configurations.

characteristic both HLS and our runtime engine exploit. The other three benchmarks were had far more regular loop structures, making the modeling of loop pipelining much simpler. In Fig. 7 we can see that GEMM had the lowest timing error of 0.21 percent. This application was the most regular of the benchmarks tested, allowing for deep pipelining across loop iterations.

To show the system-level flexibility of our simulation model, we evaluated the system performance of a multi-kernel workload across different system configurations. The workload consisted of 2D convolution, RELU activation, and max pooling kernels commonly found in deep learning applications. This workload was evaluated with four different system configurations: 1. Split kernels with distributed memory, 2. Split kernels with unified memory, 3. Unified kernel (one large kernel capturing entire workload computation), and 4. Split kernels with producer-consumer buffers. Fig. 8 presents the system cock cycles across the four system configurations. In config 1, "Distrib Mem", the host CPU was responsible for the synchronization and data transfer between accelerators. Config 2, "Unified Mem", is similar to config 1, however the host CPU was not needed for data transfer between accelerators. The unified device model of config 3 assumes that an accelerator is composed of all three kernels running sequentially. This setup offers lower control overheads for the host CPU than configs 1 and 2 by sacrificing kernel-level parallelism in multi-job workloads. In config 4, "Self Sync", synchronization and data transfer across accelerators is managed by producer-consumer buffers, without input from a centralized controller.

Overall, config 4, "Self Sync", offers lowest system clock cycles by achieving the highest degree of parallelism across kernels. All configurations are supported natively in our framework, and made available through a device configuration option in Python.[2] Some of the explored system-level configurations (e.g., "Self Sync" accelerators) are virtually impossible to be explored in the state-of-the-art simulation frameworks. "Distrib Mem" represents the only way that multi-accelerator workloads are expressed in cgem5 and gem5-Aladdin when using the SPM option, and PARADE's accelerator manager can only support Configs 1 and 2.

5 Conclusions and Future Work

This article presents an LLVM-based simulation approach for pre-RTL modeling of hardware accelerators. The proposed framework provides a cycle-level simulation model integrated into the gem5 simulation ecosystem. Computation and communication modeling are fully transparent, extensible, and parameterizable through classical object-oriented design and the intuitive Python wrappers of gem5. Our simulation environment is currently validated for timing accuracy versus an HLS-ModelSim design flow. As part of our future work, we plan to add power and area modeling, as well as support for different tightly-coupled accelerators, such as the ones proposed in [10] and [11].

2. Sample Python scripts to generate the different system configurations have been provided in our GitHub Repository.

References

[1] T. Nikolaos, K. Georgopoulos, and Y. Papaefstathiou, "A novel way to efficiently simulate complex full systems incorporating hardware accelerators," in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, Mar. 2017, pp. 658–661.

[2] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *Proc. 49th IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–12.

[3] J. Cong, Z. Fang, M. Gill, and G. Reinman, "PARADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2015, pp. 380–387.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[5] C. Menard, J. Castrillon, M. Jung, and N. Wehn, "System simulation with gem5 and systemc: The keystone for full interoperability," in *Proc. Int. Conf. Embedded Comput. Syst.: Archit. Model. Simul.*, Jul. 2017, pp. 62–69.

[6] Y. S. Shao, B. Reagan, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, 2014, pp. 97–108.

[7] J. Cong, M. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Proc. Des. Autom. Conf.*, 2012, pp. 843–849.

[8] S. Rogers and H. Tabkhi, "Locality aware memory assignment and tiling," in *Proc. 55th ACM/ESDA/IEEE Des. Autom. Conf.*, 2018, pp. 1–6.

[9] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2014, pp. 110–119.

[10] T. Nowatzki and K. Sankaralingam, "Analyzing behavior specialized acceleration," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 697–711, 2016.

[11] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman, "Needle: Leveraging program analysis to analyze and extract accelerators from whole programs," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2017, pp. 565–576.