HETSIM: Simulating Large-Scale Heterogeneous Systems using a Trace-driven, Synchronization and Dependency-Aware Framework

Subhankar Pal[*] Kuba Kaszyk[†] Siying Feng[*] Björn Franke[†] Murray Cole[†]
Michael O'Boyle[†] Trevor Mudge[*] Ronald G. Dreslinski[*]

[*]University of Michigan, USA [†]University of Edinburgh, UK
[*]{subh,fengsy,tnm,rdreslin}@umich.edu
[†]kuba.kaszyk@ed.ac.uk [†]{bfranke,mic,mob}@inf.ed.ac.uk

***Abstract*—The rising complexity of large-scale heterogeneous architectures, such as those composed of off-the-shelf processors coupled with fixed-function logic, has imposed challenges for traditional simulation methodologies. While prior work has explored trace-based simulation techniques that offer good trade-offs between simulation accuracy and speed, most such proposals are limited to simulating chip multiprocessors (CMPs) with up to hundreds of threads. There exists a gap for a framework that can flexibly and accurately model different heterogeneous systems, as well as scales to a larger number of cores.**

We implement a solution called HETSIM, a trace-driven, synchronization and dependency-aware framework for fast and accurate pre-silicon performance and power estimations for heterogeneous systems with up to *thousands* of cores. HETSIM operates in four stages: compilation, emulation, trace generation and trace replay. Given (*i*) a specification file, (*ii*) a multi-threaded implementation of the target application, and (*iii*) an architectural and power model of the target hardware, HETSIM generates performance and power estimates with no further user intervention. HETSIM distinguishes itself from existing approaches through emulation of target hardware functionality as software primitives. HETSIM is packaged with primitives that are commonplace across many accelerator designs, and the framework can easily be extended to support custom primitives.

We demonstrate the utility of HETSIM through design-space exploration on two recent target architectures: (*i*) a reconfigurable many-core accelerator, and (*ii*) a heterogeneous, domain-specific accelerator. Overall, HETSIM demonstrates simulation time speedups of 3.2×-10.4×(average 5.0×) over gem5 in syscall emulation mode, with average deviations in simulated time and power consumption of 15.1% and 10.9%, respectively. HETSIM is validated against silicon for the second target and estimates performance within a deviation of 25.5%, on average.

***Index Terms*—architectural simulation, trace-driven simulation, binary instrumentation, heterogeneous architectures**

I. INTRODUCTION

In the last few decades, there has been a strong and consistent trend of adding more parallelism into new architectures and systems [10]. Core counts have increased across all platforms, with new domain-specific accelerators being continually developed with up to thousands of cores [11], [12]. Google's TPUv3, which can scale up to 2,048 parallel cores per pod, with 32 TB of memory, is just one of many examples [13]. More recently, new architectures have been driven by a demand for accelerating not only increasingly parallel applications, but also increasingly irregular ones that rely on memory-intensive algorithms such as sparse linear algebra operations [14]–[18].

The post-Dennard scaling era has experienced a similar trend, with heterogeneous systems that have multiple CPUs working in tandem with fixed-function accelerators and GPUs [19]. This includes the broad categories of loosely-coupled accelerators, where the fixed-function logic has separate data/control path from the main pipeline (*e.g.* [20]–[22]), as well as tightly-coupled accelerators, where the logic shares resources with the pipeline (*e.g.* [23]–[26]).

Cycle-accurate simulation has been the standard tool for evaluating such systems in the pre- register-transfer level (RTL) stage, due to its level of detail and flexibility. However, this simulation approach is inherently sequential, and relies on frequent synchronization across all parts of the architecture at small timesteps. Parallel simulation efforts that use multiple event queues have not met much success, as synchronization is still needed every few timesteps [27], particularly for parallel architectures that involve shared memories. With application execution runs reaching billions or even trillions of cycles, this approach clearly does not scale well to more than a few cores, thus hitting the simulation wall [5]. In addition to the pressure on compute resources, cycle-accurate simulators are constrained by memory; these factors well limit the scale of experiments that can be performed. This is a well-established problem across the computer architecture community – benchmarks that take less than an hour to execute natively can take up to a year in detailed simulations [28].

Cycle-accurate simulation is not necessary in all steps of the design process, and can be replaced by more relaxed approaches in the early phases of design. At this stage, the design of the accelerator largely focuses on high-level features, such as the number of processing elements (PEs), sizes of caches, and relative layout of the architecture, with micro-architectural improvements requiring complete accuracy only needed in subsequent stages of design. Approximations in simulations come with a modest reduction in accuracy, however in return they greatly reduce the simulation time and memory consumption, and enable simulations of larger workloads, and a larger design space, with faster turnaround.

Prior work has explored *trace-driven* simulation techniques that offer good trade-offs between simulation accuracy and speed, however, there exists a gap for a framework that enables fast simulation of large-scale heterogeneous systems with many PEs, while providing reasonably accurate estimations

TABLE I
COMPARISON OF HETSIM WITH PRIOR TRACE-DRIVEN SIMULATION FRAMEWORKS.

Work	ISA	Threading	Execution	Sim. Limit	Synchro-nization	Target Type	Trace Gen./ Replay	Reported Sim. Speed	Reported Timing Error
SST/Macro [1]	N/A	Multi	OoO	1,000	MPI	Multi-CMP Systems	Native/Custom	N/R	<10.0%
Netrace [2]	Agnostic	Multi	InO	64	N/A	CMPs	gem5/Custom	N/R	N/R
Elastic Traces [3]	Agnostic	Single	OoO	N/A	N/A	CMP	gem5/gem5	7.2× over gem5-FS	15.3%
ElasticSim-MATE [4]	Armv7/8	Multi	OoO	128	OpenMP	CMP	gem5/gem5	3.8× over gem5-FS	7.1%
Synchro-Trace [5]	Agnostic	Multi	InO	64	Pthreads/ OpenMP	CMP	Native/gem5	9.6× over gem5-FS	5.7%
AccelSim [6]	SASS/PTX	Multi (Warp)	InO	80	Cycle-Level	NVIDIA GPU	Nvbit [7]/Custom	12.5 KIPS	15.0%
MacSim [8]	PTX/GEN	Multi (Warp)	InO	48	Cycle-Level	NVIDIA/Intel GPU	PTX/ProtoBuf	N/R	N/R
Rhythm [9]	x86	Multi	InO/OoO	>32	Pthreads	CMPs	Native/Native	O(MIPS)	7.2%
HETSIM [this work]	**Agnostic**	**Multi**	**InO with OoO memory accesses**	**4,160**	**Pthreads/ Custom**	**Accelerators/ Heterogeneous hardware**	**Native/gem5**	**5.0× over gem5-SE ∼10× over gem5-FS**	**15.1% over gem5-SE 25.5% over silicon**

of important metrics. We observe that the traces used in many existing frameworks are fairly generic, with at least three key features missing from them:

1) Tokens defining inter-PE communication.
2) Annotations of dependent memory addresses.
3) Program counters, in case of programmable PEs.

Tokens that encode communication between heterogeneous PEs are critical in ensuring that the performance impact of communicating between PEs is modeled correctly, *e.g.* the performance difference between modeling a blocking call and omitting it could even be equivalent to the runtime of a full application on a particular PE. Furthermore, the number of units executing concurrently at any point will have a significant impact on the estimated power of the system.

Dependency analysis is critical when modeling accelerators, in particular for those that target memory-bound applications and support memory-level parallelism [29]. Within our tracing framework, we employ strategies that trade-off reduced accuracy for compute operations, in return for improved accuracy in modeling the PEs' interaction with the memory subsystem.

Prefetching is a well-known technique to ensure that data is available in the caches when the program requires it, and often relies on the program counter [30], [31]. Tracking the program counter in our traces is a simple, yet important addition when attempting to accurately model the memory subsystem.

As a solution to fill the gap in prior work, we have developed HETSIM, an end-to-end framework that targets new or existing users of gem5, which is a widely-used system and architectural simulator [32], [33]. We expose significant features missing from existing trace formats, that allow us to accurately model modern applications and target hardware including tokens defining heterogeneous communication, dependency tracking between memory accesses, and tracking the program counter. Furthermore, by trading-off detail where it is not critical, HETSIM improves simulation times by 3.2×-10.4×, with little additional overhead.

We provide users with a flexible, unified infrastructure, containing utilities for generating enhanced traces, replaying them, and incorporating new architectural features. For this, we introduce the notion of a **primitive**, which in this context is a hardware feature that can be modeled in software emulation, *e.g.* an operation where a PE interacts with a hardware buffer. HETSIM is packaged with a set of such common primitives, and also allows the user to add support for custom primitives. Additionally, each primitive has an entry in a *trace specification* file that contains metadata information required to model it accurately during simulation. In Section IV, we demonstrate the required efforts for multiple use-cases, comprising of (but not limited to) using HETSIM with a new model, profiling new applications, tweaking an existing model, and so on. HETSIM additionally features a standalone mode, that is useful for rapid early-stage evaluation without building detailed PE models.

We evaluate HETSIM on two heterogeneous target architectures – a recently-proposed programmable, reconfigurable accelerator, and a fixed-function ASIC for sparse matrix-matrix multiplication (SpMM). Across a set of three well-known workloads, HETSIM achieves runtime and power estimations errors of 0.2%-57.0% (average 15.1%) and 0.0%-24.2% (average 10.9%), respectively, while achieving simulation time improvements of 3.2×-10.4× (average 5.0×), over a detailed gem5 model in syscall emulation mode. HETSIM is also validated against prototyped silicon, and estimates runtime with a deviation of 2.2%-46.4% (average 25.5%) of the chip.

II. RELATED WORK

Trace-Driven Simulation. A few prior work have explored techniques that enable faster simulations by relaxing the constraints of full-system simulation. However, they are targeted specifically for general-purpose processors, such as CMPs and GPUs. We provide a qualitative comparison of HETSIM against these work in Table I.

Perhaps the closest to our work is SynchroTrace [5], [34]. Like SynchroTrace, HETSIM is a dependency- and synchronization-aware trace-based simulator that uses native execution for trace generation and gem5 for trace replay. However, SynchroTrace targets CMP systems and their frontend tools work with standard Pthreads-based multithreaded code. In contrast, HETSIM targets accelerators and heterogeneous architectures and uses the notion of primitives defined in a trace specification format to model the behavior of custom hardware blocks in the target. Unlike SynchroTrace, which works with unmodified benchmarks, HETSIM requires the user to implement *parts* of their multithreaded code using HETSIM primitives. However, this requirement is key in enabling HETSIM to model arbitrary hardware functionality beyond CMPs.

Heterogeneous System Simulators. gem5-Aladdin [35], based on the Aladdin pre-RTL estimation framework [36], provides a gem5-based infrastructure that explores the design

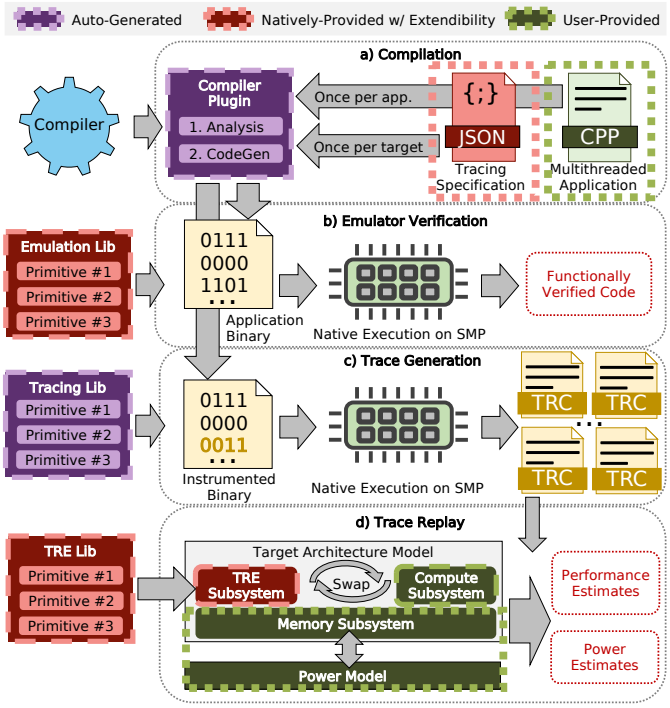


Fig. 1. The end-to-end approach employed in HETSIM, consisting of four stages: compilation, emulator verification, trace generation and trace replay. In its current implementation, HETSIM uses LLVM as the compiler on the native machine, and gem5 for target architecture modeling and trace replay.

space of fixed-function accelerators and captures their interactions within the SoC. Although gem5-Aladdin explores DMA interactions in the SoC, Aladdin only focuses on standalone datapaths and local memories. HETSIM, in contrast, focuses on simulating a single target accelerator and faithfully modeling its interaction with the memory subsystem. Rogers *et al.* recently proposed a gem5 extension for accelerator modeling that uses an LLVM frontend for cycle-accurate full-system simulation [37]. HETSIM, on the other hand, performs trace-driven simulations and uses LLVM only as a frontend to instrument the emulation binary with trace-emitting function calls. PARADE is a cycle-level full-system simulator for the design-space exploration of accelerator-rich systems, and is also integrated into gem5 [38]. It uses high-level synthesis (HLS) and RTL simulation for the accelerator, and gem5 for the uncore [39]. HETSIM uses simple trace replay engines that parse through trace files and execute the trace operations, while using native gem5 support for the rest of the system. ZSim is another well-known simulator that is an alternative to gem5 and scales up to thousands of cores [40]. In contrast to HETSIM, however, ZSim targets heterogeneous systems consisting of general-purpose cores only.

To the best of our knowledge at the time of writing, HETSIM *is the first customizable trace-driven simulator targeted for heterogeneous accelerator-based systems, that enables fast early-stage design-space exploration of novel architectures.*

III. PROPOSED APPROACH

HETSIM comprises a streamlined multi-phase simulation infrastructure that aims to remove much of the burden from the user, and enable larger simulations with reduced overheads, while incurring little loss of accuracy in terms of both performance and power estimations. At a high level, HETSIM follows a *trace-driven* approach; execution traces for an application are generated from an emulation of the target architecture on a native machine. This is followed by a replay of the traces on a lightweight model of the target architecture that swaps out the compute units, or processing elements (PEs), in the target with trace replay engines (TREs). This is illustrated in Figure 1 and is discussed in-depth in the remainder of this section.

HETSIM requires the following inputs from the user:

- A target architectural model that comprises a set of parallel PEs and a memory subsystem. Some examples of such architectural templates supported by HETSIM are: multi-core CPUs (PE = thread), systolic arrays (PE = multiply-accumulate unit + control logic), coarse-grained reconfigurable architectures (PE = ALU + routing logic), and so on. It is assumed that target-specific hardware interactions, such as a PE pushing data to its neighbor, are already handled in the detailed model provided by the user. These interactions are mapped to HETSIM primitives, and while HETSIM provides a library of common primitives, it can also be extended to support custom ones.
- A power model of the target architecture. HETSIM supports any arbitrary power modeling tool that is compatible with gem5, as long as the tool can be enhanced to extract PE activity from the HETSIM-generated traces or TRE statistics.
- A C/C++ program that models each PE using Pthreads and invokes *primitive* implementations from HETSIM's emulation library. HETSIM flexibly supports the addition of new primitives (discussed in Section IV-D2).
- A tracing specification for primitives in the target hardware. This is implemented as a JSON file with one entry per primitive operation, and allows the user to tune knobs, such as latency, that model the behavior of the primitive when executed by a given PE-type on the target architecture.
- A native SMP machine with sufficient threads and memory capacity for trace generation and replay. A system with more resources speeds up the trace generation phase, although trace generation overhead is generally amortized by reusing the traces across multiple experiments.

HETSIM also supports a *standalone mode*, in which the user builds their gem5 model around the TREs to begin with, rather than providing a model consisting of detailed PEs.

The framework is split into four distinct, yet fully integrated stages: compilation, emulator (functional) verification, trace generation, and trace replay. These are illustrated in Figure 1. HETSIM supports heterogeneous PEs that differ in terms of sequence of operations, latency of each operation, among other factors. In its current state, HETSIM uses gem5 for trace replay, and supports the syscall emulation (SE) mode.

Functional Correctness. HETSIM takes a functional-first approach and validates the output of the user application during the emulator verification stage. The results of simulation, *i.e.* the trace replay step, however, are not functionally correct, since HETSIM traces do away with the original compute op-

TABLE II
LIST OF PRIMITIVES BUILT INTO HETSIM. IMPLICIT AND EXPLICIT PRIMITIVES ARE DISTINGUISHED BY @ AND __ PREFIXES, RESPECTIVELY.

Primitive	Description
`@load(&a)`	Load word from address `&a`
`@store(&a,v)`	Store word `v` to address `&a`
`@iop`	integer op; modeled as stall during replay
`@fop`	floating-point op; modeled as stall during replay
`__load_block(&a)`	Load word from address `&a` and block other primitives until response returns
`__load_uncache(&a)`	Load word from address `&a` directly from main memory
`__load_block_uncache (&a)`	Load word from address `&a` directly from main memory and block other primitives until response returns
`__store_block(&a,v)`	Store word `v` to address `&a` and block other primitives until response returns
`__store_uncache(&a,v)`	Store word `v` to address `&a` directly into main memory
`__store_block_uncache (&a,v)`	Store word `v` to address `&a` directly into main memory and block other primitives until response returns
`__barrier_init(&b,n)`	Initialize barrier at address `&b` with `n` PEs
`__barrier_wait(&b)`	Wait at barrier at address `&b`
`__mutex_lock(&m)`	Lock mutex at location `&m`
`__mutex_unlock(&m)`	Release mutex at location `&m`
`__sleep()`	Go to sleep until signaled by another PE
`__signal(id)`	Signal PE[`id`] to wake up
`__push(dir,v)`	Push `v` to PE in direction `dir`
`__pop(dir)`	Pop from PE in direction `dir`
`__dump_stats()`	Dump gem5 statistics into `stats.txt`
`__reset_stats()`	Reset gem5 statistics
`__dump_reset_stats()`	Dump gem5 statistics into `stats.txt` and reset them

erations. Instead, HETSIM attempts to model the performance (and optionally, power) of the target as accurately as possible.

A. Creating the Tracing Compiler Plugin

The tracing plugin (Figure 1-a) contains the compiler pass that automatically instruments the user's application with tracing calls. It is an LLVM plugin that is invoked as a user-defined pass in the `clang` compiler. It is generated once per target architecture from a user-specified list of primitives, which are extracted from a *trace specification file*. The specification file is composed of a list of primitives and their mappings to trace tokens, augmented with metadata necessary for instrumentation and accurate modeling of the primitive. Additionally, the specification includes valid address ranges for the target, in order to filter-in only relevant memory accesses. Primitives to be traced can be user-specified as an allow-list of valid instruction types, or a deny-list of ignored primitive types. Further filtering can be done in the application source code, by specifying regions-of-interest (ROIs) to be traced.

Two types of primitives can be traced - *explicit* and *implicit*. Explicit primitives include any primitives that are explicitly called in the emulated application. These for example, can include synchronization primitives, such as lock or unlock calls on a mutex. Implicit operations include any primitives that are identified by the compiler without user specification, such as compute operations (modeled as stalls) and memory accesses. These constructs are not represented in the emulated code by any specific function calls, but are implicitly performed, for example by accessing arrays or performing arithmetic operations, however they are critical to the performance prediction, and therefore need to be traced. These operations are instead traced at the LLVM intermediate representation (IR) level, and are specified by instruction type.

HETSIM is provided with a trace specification containing primitive constructs (see Table II) that are common across many computing paradigms, such as synchronization primitives (CPUs and GPUs), queuing and signalling operations (domain-specific accelerators such as TPUs), and so on. However, the system is designed to be flexible, and the trace specification is exposed to the user to allow the addition of *custom primitives*. The format for this specification has been designed to have low overhead, requiring only the bare minimum input from the user. The format is a JSON file, and contains the following components:

1) The function name or IR instruction that is to be traced.
2) The trace token emitted into the trace when the function is encountered.
3) Optional arguments to the trace token, and their mappings, *e.g.* address and value for a load operation.
4) The latency (cycles) that this token is emulating.
5) Memory access penalties, designed to mimic the overhead of dynamic operations such as synchronization.

B. Compilation and Emulator Verification

In this step, the user provides the application program that is to be simulated on the target. The compiler generates two binaries. The first is un-instrumented, and is used for functional verification of the program (Figure 1-b). The second, is an instrumented binary, and is used for trace generation (Figure 1-c). When creating the instrumented binary, the compiler takes the application source, scans it for uses of functions described in the tracing specification, and instruments these functions with the appropriate tracing calls. A use case and specification format are discussed later in Section IV.

The emulation code is automatically instrumented to generate traces using the LLVM compiler plugin. The emulation code is first compiled into LLVM IR, after which our custom compiler pass scans the IR for occurrences of accelerator primitives, and injects a tracing call for each primitive. The tracing call holds references to any relevant parameters, which are resolved later during runtime. The resulting binaries are compiled for the native SMP, however our tracing infrastructure relies only on functional behavior, which remains consistent across all ISAs.

Following the compilation phase, HETSIM performs functional verification of the program, by executing it natively.

1) Memory Operations: To instrument memory operations, the compiler scans for load and store operations and injects tracing calls into the code. During runtime, these calls generate tokens into the trace file. The token entry for a memory access records whether it is a load or store, the virtual program counter at which it occurs, and a list of dependent loads and stores. This list preserves operations that must be executed before the current operation, and is useful for modeling targets with wide in-order cores or PEs that support multiple outstanding memory accesses. We rely on LLVM's data-dependence graph (DDG) [41], [42], in order to add dependence information to each memory token.

2) Arithmetic and Miscellaneous Operations: Upon encountering any non-memory instruction types, the compiler increments the number of cycles associated with that specific

instruction, *i.e.* stalls. Typically, these would include integer and floating-point compute operations, which are a crucial part of any application code. To avoid runtime overhead, the compiler only emits these when absolutely necessary. We have identified two points at which the compiler must emit any outstanding stalls. Firstly, on encountering a memory instruction, in order to preserve the correct ordering of arithmetic and memory operations. Secondly, at the end of a basic block – since this compiler pass is static, we cannot increment the cycle count across basic block boundaries, as we do not know for certain how dynamic control flow will execute.

Noting the shift in emerging applications from compute-dominated ones to those bounded by memory, we deem that tracing all non-memory instructions may result in unnecessary slowdown, without necessarily improving accuracy. Therefore, in order to allow flexibility, we allow the user to specify which instructions are to be traced. This can occur either via allow-listing them in the specification file for the compiler plugin, or deny-listing instructions that do not require tracing.

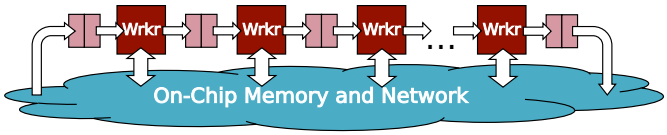
C. Trace Generation

Trace generation (Figure 1-c) is performed with the instrumented application binary. During execution, instrumentation calls are made to the tracing library, which emits trace tokens into the trace file(s). Since this is an emulation of the target system on a native SMP, the runtime system plays an important role in performing the mapping from host to target. For example, the compiler alone cannot resolve whether a memory access is made to target memory or to host memory. The compiler therefore instruments all memory operations within the marked code segment, and the runtime decides whether or not the accessed address is a valid access to the target memory, based on information in the trace specification. If it is, then a tracing call is executed by the runtime. Otherwise, the access is executed for functional correctness, but ignored in the trace.

Manual Trace Generation. The tracing API is exposed, should the user wish to manually insert tracing calls into the application program. This can be useful in situations where more fine-granular performance tuning is required, for example, where different instances of the same primitive type, executed by the same PE, incur different latencies in the target architecture. Another instance could be for a target executing a multi-phase algorithm, where the simulator needs to be artificially throttled to emulate data transfer latency across algorithm phases. We note that this is orthogonal to automatic tracing and only required to model pathological cases.

D. Trace Replay

The final step in the execution of HETSIM is trace replay (Figure 1-d). As a prerequisite to this step, the user's detailed gem5 model is modified to swap out detailed core models for our trace replay engines (TREs). A TRE is a simple engine that parses through its designated trace file and executes each trace token. For simple operations, such as stalls, the TRE simply schedules to read the next trace token after "stall_count" cycles. This simplification of convoluted operations is the most important source of speedup for HETSIM. Memory operations *i.e.* loads and stores, in contrast, are sent out as packets to the memory subsystem in the same way as with the detailed model. This leads to high fidelity of the activity in the on-chip network and memory, as well as off-chip, but trades-off simulation speed.



a) Example target architecture: 1D systolic array

b) Application Code

```
1: [...]
2: __register_core_id(peId); // populate internal map
3: pthread_barrier_t bar;
4: // assume bar is initialized to N_PE (number of PEs)
5: [...]
6: __barrier_wait(&bar); // uses emulation library
7: // iterate over array of size N
8: for(int i=0; i<N; ++i) {
9:   float v;
10:  if(peId == 0) v=A[i]; else val = __pop(Dir::LEFT);
11:  v = foo(peId, val);
12:  if(peId != N_PE-1) __push(Dir::RIGHT, v); else A[i]=v;
13: }
14: __barrier_wait(&bar);
15: [...]
```

c) Emulation Library

```
1: [...]
2: // implementation of blocking push using STL queue
3: void __push(Dir dir, float v) {
4:   auto myID = tid_to_core_id_map.at // internal map
5:       (std::this_thread::get_id());
5:   // invoke helper that returns systolic queue ID
6:   q_intfc_t * queue = queues.at(getQID(myID, dir));
7:   std::queue<uint64_t> & q = queue->q;
8:   std::mutex * m = queue->m;
9:   std::condition_variable * emptyCv = queue->emptyCv,
10:     * fullCv = queue->fullCv;
11:  // grab the lock to this queue
12:  std::unique_lock<mutex> lock(*m);
13:  while (q.size() == queue->size) { // check if full
14:    // release lock and sleep until consumer PE notifies
15:    fullCv->wait(lock);
16:  }
17:  q.push(v);
18:  // if some PE is waiting on this queue, notify them
19:  empty->notify_one();
20: }
21: [...]
```

Fig. 2. a) Example accelerator composed of 1D systolic array of worker PEs with datapath to main memory via an on-chip network. b) Example multithreaded application program written in `C++` with `std::threads` that uses HETSIM primitives (prefixed with "__"). c) Implementation of one of these primitives, *i.e.* `__push()`, as part of the HETSIM emulation library.

The actual trace replay step entails running the traces generated in the previous step through the TRE-enabled gem5 model. At the end of the replay step, statistics are dumped as with any other gem5 simulations. While statistics associated with the detailed core model are lost, the TRE model itself provides some basic statistics, such as the load count, store count, stall count, and "instructions" completed. Finally, the optional power model provided by the user is modified to parse through these statistics (or the generated traces) to analyze activity at the PE-level, in addition to estimating the power connsumed by the rest of the system.

IV. USAGE

We describe in detail various use cases of HETSIM in this section, with the example of a target architecture.

A. Supporting a New Architectural Model

A new architecture is implemented in HETSIM using the following end-to-end steps, with automated steps which other-

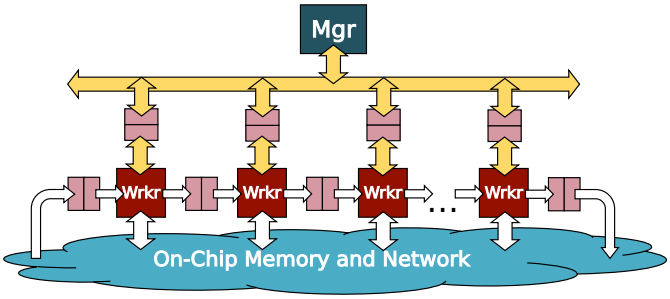


Fig. 3. An enhanced version of the target architecture in Figure 2 a). A manager PE is added that communicates over a bus to the worker PEs, using hardware FIFO buffers.

wise would require significant additional user effort italicized:

1) Implement the detailed model in gem5, along with a power model if power estimations are desired.
2) For each primitive in the trace specification, tweak primitive parameters (per PE-type) to match its behavior on the target. HETSIM *automatically generates an updated compiler plugin based on the specification file.*
3) Modify the gem5 model to instantiate TRE objects in place of the detailed PE models.
4) Modify the power model to incorporate power estimation of the PEs based on corresponding TRE statistics or traces.
5) Implement the application to be profiled using primitives from the emulation library. Note that if the design requires custom primitives, they are incorporated as discussed below in Section IV-D2.
6) *Run the* HETSIM *compiler pass to generate the instrumented and un-instrumented application binaries.*
7) Execute the un-instrumented binary on the native machine and verify that the code is functionally correct.
8) Execute the instrumented binary on the native machine to generate the traces.
9) Run the TRE-enabled gem5 model to obtain performance and power estimates with the generated traces.

Standalone Mode. In the standalone mode, the user simply skips implementing detailed models of the PEs in the target architecture (*i.e.* in step 1 above) and instead uses the TREs directly in their model. This is useful at early stages of the design process, when coarse performance/power estimates are desired with fast turnaround times.

Illustrative Example. We consider a target architecture composed of a simple 1D systolic array [43] of PEs that are connected via hardware FIFO queues (Figure 2-a). Here, the PEs have access to the main memory via an on-chip network. Figure 2-b presents an application program that runs on this architecture. Worker[0], the first PE, streams-in an array `A` from the memory into the systolic array, element-by-element. Each worker applies a function `foo()` to the element and passes to its right neighbor. The final worker writes the result back to array `A` (in-place). We note that this example uses three primitives, namely `__barrier_wait()`, `__pop()`, and `__push()`, that have target-specific hardware implementations and are *not* software constructs on the target. `__register_core_id()` is a generic primitive call required for emulation runs in HETSIM. These calls are implemented as part of HETSIM's emulation library and invoked while running the steps in Sections III-B and III-C. We illustrate, in Figure 2-c, the emulation library implementation of one of these primitives, `__push()`, which models the behavior of a worker PE pushing to a neighboring FIFO (and blocking if the FIFO is full). This specific implementation uses locks to ensure mutually-exclusive accesses to the FIFO by two PEs, and condition variables to avoid busy-waiting, thus speeding up the trace generation step. Note that the primitives used in this example are not supported by existing trace-driven simulation frameworks, and yet they are commonplace in heterogeneous accelerators, We posit that it is critical to simulate such primitives correctly for accurate estimations.

B. Profiling New Applications on Existing Models

Another typical use of HETSIM is when new applications are executed on an existing architectural model. Consider again the system in Figure 2-a. Say that have a new application for the systolic array that operates on a matrix `M` instead of array `A` in L10. The user first implements the changes on top of the existing application code, *e.g.* adding new `for`-loops, bounds-checking, *etc.* Next, steps 6-9 outlined above are repeated. Without an automated tracing system, the user would need to manually instrument the application source code, which is both time-consuming and prone to errors.

C. Exploring New Design Points on an Existing Model

Given the TRE-enabled architectural model, a power model, and an application to profile, HETSIM easily allows users to perform design space exploration on the target hardware.

In our running example, the user may choose to, say, explore power-performance trade-offs for the following:

- Sizing the FIFO structures in the design.
- Sizing bus widths and other interconnect parameters, such as the number of virtual channels [44].
- Clock speed and dynamic voltage-frequency scaling.
- Capacities of caches in different levels of hierarchy.
- Experiments with scaling-out the number of cores.

Note that the user only needs to re-run the trace generation step when they want to explore design-points that affect the operations executed by a PE. Among the listed experiments, only the final one (core-scaling sweeps) requires trace generation to be repeated for each point.

D. Modifying the Hardware on an Existing Model

Next, we describe the steps to use HETSIM with modifications to an existing TRE-enabled architectural model. We now add heterogeneity to our running example by instantiating a "manager PE" in the design that communicates with the worker PEs via FIFO buffers (Figure 3). The user thus provides HETSIM with an updated gem5 model and power model.

1) Extension with Supported Primitives: Say that we want to now use the manager PEs to distribute pointers to vector `A` before the start of computation. The user writes a new program for the manager PE that includes `__push()` calls corresponding to each worker PE. Also, the existing worker PE application code is modified to include the corresponding `__pop()` call. Note that this modification leverages existing HETSIM primitives. After this, steps 6-9 in Section IV-A are re-performed, and no additional steps are required.

a) User Spec

```
[...]
"pe" : { // contains entries for each PE type
  "mgr" : { // contains primitives for PE type == "mgr"
    [...]
    "__push_bcast(float)" : { // primitive_name(arg)
      "token": "PUSH_BCAST", "cycles": 2, "enable": 1
    },
[...]
```

Auto-Generate

b) Tracing Library

```
[...]
void __trace__mgr__push_bcast(float v) {
  register_trace_entry("PUSH_BCAST " +
    std::to_string(cycles) + "\n");
}
[...]
```

c) Emulation Library

```
[...]
void __push_bcast(float v) {
  for (int i=0; i<N_PE; ++i) {
    __push(Dir::PE + i, v); // reuse existing primitive
  }
}
[...]
```

d) TRE Library

```
[...]
void TRE::tick() { // this is called each "clock tick"
  [...]
  else if (getNextToken() == "PUSH_BCAST") {
    // fill metadata into global object bcast
    bcast.blocked = false;
    // get cycle penalty
    bcast.cycles = std::stol(getNextToken());
    bcast.source = this;
    schedule(pushBcastEvent,
             clockEdge(Cycles(bcast.cycles)));
    [...]
}
void TRE::pushBcast() {
  // iterate over all TREs, skipping over the current one
  for (auto &tre : allTREs) {
    if (tre != this) {
      // check the queue that connects this to consumer
      if (queues->q.at(this, tre).size() ==
          queues->size) {
        // give up for now if the queue is full
        bcast.blocked = true; return;
      }
    }
  }
  // all consumers' queues are !full, so push to all
  for (auto &tre : allTREs) {
    if (tre != this) queues->q.at(this, tre)
      .push(0); // value doesn't matter
  }
  schedule(tickEvent, clockEdge(interval)); // continue
}
void TRE::pop() {
  [...]
  // if push was blocked because of full queue(s), release
  if (bcast.blocked) {
    bcast.source->schedule(pushBcastEvent,
         clockEdge(Cycles(bcast.cycles)));
    bcast.blocked = false;
  [...]
}
```

Fig. 4. Example steps to add support for a new primitive, `__push_bcast()`, that broadcasts a piece of data from one PE to all others. a) The user augments the specification file with the new primitive. Here, the user enables the primitive and specifies its metadata only for the manager PE-type. b) The user specification file is automatically parsed into the tracing library call. c) The user extends the emulation library with an implementation for the new primitive. d) The user models the behavior of the new primitive by extending gem5's TRE class implementation.

2) Extension Requiring New Custom Primitives: Consider the case where the user wants to optimize the hardware to push a task to *all* worker PEs using a single instruction. The user first implements hardware support in the original gem5 model to enable broadcast in the manager ↔ worker bus. Next, the following extensions to HETSIM are made (Figure 4):

- Figure 4 a): A new entry in the trace specification file, `__push_bcast()`, is created with appropriate metadata.
- Figure 4 b): HETSIM automatically generates an updated compiler plugin based on the specification file. The implementation is shown in the figure.
- Figure 4 c): The user creates an implementation of the new primitive in the emulation library source code.
- Figure 4 d): Finally, the user implements the behavior of the new primitive in the TRE source code (part of gem5) and then re-builds gem5.

Lastly, steps 6-9 discussed in Section IV-A are re-executed to obtain estimations from HETSIM for the modified target.

a) Manager Traces

```
[...]
STALL 100 ( ) // startup latency
PUSH_BCAST 2  // stall 2 cycles &
// then broadcast to all workers
PUSH_BCAST 2
PUSH_BCAST 2
STALL 1 ( ) // empty list
              // indicates no deps
POP 1 0 // PE-ID, no extra stalls
POP 2 0
POP 3 0
[...]
```

b) Worker Traces

```
[...]
STALL 100 ( ) // startup latency
POP 0 2 // mgr ID on mgr<->wrkr bus
POP 0 2
POP 0 2
LD @10 0x2000 ( ) // A[0]
LD @11 0x3000 ( ) // B[0]
STALL 1 ( 0x2000 0x3000 )
                  // compute A[0]+B[0]
ST @12 0x4000 ( ) // C[0]=A[0]+B[0]
PUSH 0 0 // mgr ID on mgr<->wrkr bus
[...]
```

Fig. 5. Snippets of the trace generated for an example application on the target hardware in Figure 3. The manager broadcasts pointers to three arrays, `A`, `B` and `C`, to worker PEs. The workers collaboratively perform a vector-sum of `A` and `B` into `C`. Each memory access token is annotated with the virtual PC (prefixed with @). The arrows indicate dependencies captured between the loads and the addition operation. One key feature of HETSIM is that it enables modeling multiple outstanding accesses by tracking these dependencies. Note that the comments are not emitted in the actual traces.

E. Comparison with different PE Types

Users may need to perform studies of different PE types with varying levels of heterogeneity in the architecture. This is natively supported in HETSIM. For instance, with the `__push_bcast()` primitive in our running example, we can simply vary the the `cycles` parameter in the specification file to model different types of manager cores. A similar modification to the power model is also required (not shown).

F. Modeling ISA Extensions

HETSIM also supports ISA extensions for any type of processor. The extension itself needs to be defined as another primitive, and incorporated into HETSIM as explained in Section IV-D2. HETSIM does not distinguish between ISA extensions and hardware primitives.

G. Example Traces

The generated traces for the manager and worker[0], for the example of a vector-addition application, is shown in Figure 5. HETSIM preserves dependent addresses for operations that do not block, *e.g.* loads, stores, and stalls. In Figure 5 (right), we see that worker[0] can concurrently issue loads to elements `A[0]` and `B[0]`, assuming that the worker PE itself is capable of issuing multiple memory accesses. This accurate modeling of the behavior of complex PEs is made feasible by HETSIM's dependency tracking feature.

V. EXPERIMENTAL SETUP

We evaluated HETSIM on an AMD Ryzen Threadripper 2990WX 32-core processor with 2-way SMT, 128 GB 2133 MHz DDR4 main memory, and 2 TB PCIe SSD storage. Our experiments were performed on Ubuntu 16.04, however HETSIM can be used with any operating system supported by

gem5. HETSIM requires LLVM (minimum version 10.0) for the tracing compiler support. With this setup, we were able to simulate a system with up to 4,160 PEs, beyond which the memory capacity became the bottleneck.

We use HETSIM to perform design space exploration on two target accelerator systems.

1) Transmuter Architecture: Our first target for evaluation using HETSIM is a recently-proposed reconfigurable accelerator called Transmuter [45]–[47]. Transmuter is highly amenable as a target architecture for HETSIM, since it exposes multiple options for both design-time parameter exploration and run-time reconfiguration.

Design. Transmuter is a tiled heterogeneous architecture composed of a set of in-order, general-purpose processing elements (GPEs), with a separate local-control processor (LCP) that manages a tile of GPEs. See Figure 4 in [45] for a detailed block diagram. The GPEs are connected through a two-level (L1 and L2), non-coherent, reconfigurable cache-crossbar hierarchy to a high-bandwidth memory interface (HBM). The reconfigurable network and memories enable Transmuter to morph into one of sixty-four possible configurations.

Workloads. We evaluate Transmuter with HETSIM on three important linear algebra applications, namely general matrix-matrix multiplication (GeMM), general matrix-vector multiplication (GeMV) and sparse matrix-matrix multiplication (SpMM). These workloads exhibit varying characteristics in terms of the nature of data (dense/sparse) as well as arithmetic intensities (compute-/memory-bound).

GeMM and GeMV are implemented using traditional blocking/tiling techniques that are deployed in modern software libraries [48], [49]. The input matrix is blocked to fit into the L1 cache of the target design. The SpMM implementation is based on the outer product algorithm where the SpMM is performed in two phases, the *multiply* and *merge* phases [17].

Modeling and Correlation. The detailed reference Transmuter model and power calculator are taken from the prior work [45]. Transmuter maintains its input and output data structures in a memory region that is separated from each core's stack/heap, called SHARED_SPACE. We allow-listed only memory accesses to this region in HETSIM to achieve higher simulation speeds. For the detailed (baseline) model, we instantiated the GPEs and LCPs as scalar, single-issue MinorCPU cores. We also adapted the power model to estimate core power using the statistics dumped by TREs.

For our experiments with HETSIM, we switched the MinorCPU cores with our TREs that replay the traces generated from the instrumented application binary (Section III), and removed the instruction caches. We correlate the performance and power estimations provided by HETSIM with the detailed gem5 model in SE mode for the three workloads.

Note that we had to extend HETSIM with primitives that are specific to the Transmuter architecture. Particularly, we implemented the `__pop_mmap()` and `__push_mmap()` primitives on top of the existing `__push()` and `__pop()` calls. These implement the push and pop functionalities, respectively, as loads and stores to special addresses that access the memory-mapped work-queue and status-queue hardware structures in Transmuter.

Design Space. We sweep the following design points in Transmuter, and use HETSIM to perform the experiments.

1) *Design-Time.* Number of tiles, number of GPEs per tile, L1 bank sizes, L2 bank sizes, off-chip bandwidth.
2) *Run-Time (Hardware).* Reconfiguration of the L1 and L2 caches from private to shared and vice versa.
3) *Run-Time (Software).* Block size parameter in the application code for GeMM and GeMV.

Owing to space constraints, we omit presenting our experiments with core count and memory bandwidth scaling on Transmuter in this paper.

2) SpMM Accelerating Chip: We now discuss the case study of a prior heterogeneous sparse matrix-matrix multiplication (SpMM) accelerator prototype chip [18], [50].

Design. This chip accelerates the outer product algorithm using a loosely-coupled heterogeneous accelerator design. The chip is composed of 8 tiles with 4 fixed-function multiplier PEs and 2 in-order Arm Cortex-M class cores per tile. See Figure 4 in [18] for a block diagram. A PE comprises a floating-point multiplier, an 8-wide outstanding-request queue that tracks loads and stores to the memory system, and simple control logic. The in-order cores in a tile are a pair of Arm Cortex-M0 and Cortex-M4F processors. They share ports to an L0 crossbar with the PEs. Each tile has an L0 cache layer and the tiles share a second (L1) layer. Further, the L0 cache can reconfigure between cache and scratchpad memory. Lastly, the chip communicates with an off-chip host through a custom front-side bus interface.

Workload. The multiply phase in the chip is performed by the PEs with the L0 in cache mode. For merge, the chip leverages the decoupled access-execute paradigm (DAE) [51] to partition the outer product merge phase computation between a fetch core (Cortex-M0) and a merge core (Cortex-M4F). During the merge phase, the M0 prefetches data into the L0 scratchpad, while the M4F sorts and merges lists of partial products in the L0 scratchpad. The Arm cores are turned off during the multiply phase and the PEs are turned off during merge.

Modeling and Correlation. We considered a HETSIM model based on the chip specifications (32 PEs, 8 merge cores, 2 kB cache banks, *etc.*) and evaluated a multithreaded version of the outer product application code. We instantiated two PE types in the user specification to model the differences between the fixed-function PE and the Arm core. Dependency tracking in HETSIM was particularly beneficial to faithfully model the multiple outstanding requests supported by the PEs, which is critical for the memory-bound SpMM workload. In addition, HETSIM's capability of modeling heterogeneous PEs allowed for it to effectively model DAE in the merge phase.

VI. EVALUATION

In this section, we present details on our experiments with HETSIM and comparison with detailed gem5 models for the two target accelerators in Section V.

A. Simulation Time Profile and Scalability of HETSIM

Figure 6 shows the wall clock time for trace generation and replay on the native SMP for two problem sizes across each of the evaluated workloads. Except for SpMM, the overhead of

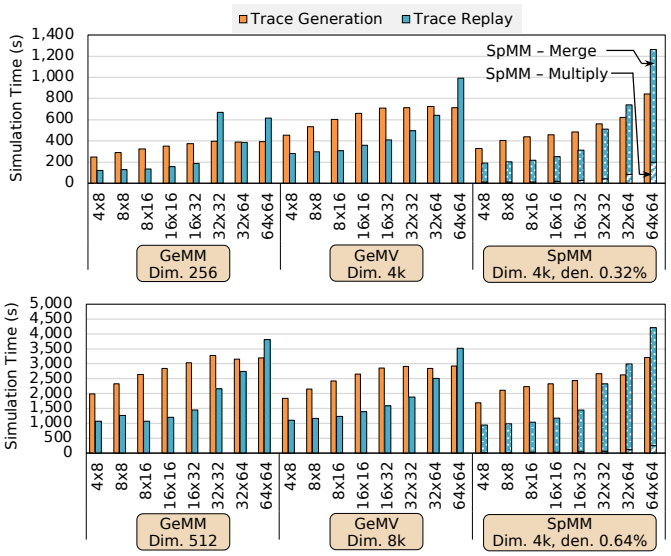


Fig. 6. Trace generation and replay timing profile on the Threadripper system, across different workloads and Transmuter sizes. Each simulated core is a 4-stage in-order pipeline. The total number of cores simulated for an $N_T \times N_G$ Transmuter configuration (x-axis) is $N_T \cdot (N_G + 1)$, where N_T is the number of tiles and N_G is the number of GPEs per tile.

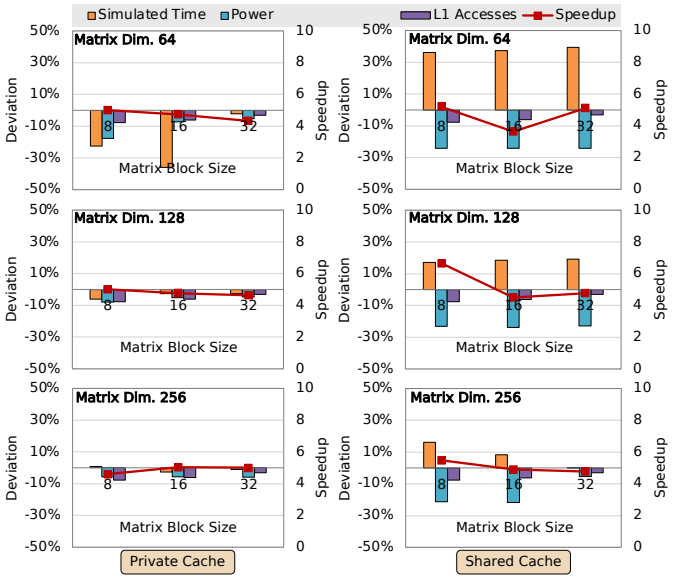


Fig. 7. Accuracy of performance, power and L1 cache access count estimations of HETSIM compared to the detailed gem5 model of 2×8 Transmuter, for GeMM with varying block sizes and two HETSIM configurations.

(parallelized) trace generation plateaus with increasing number of simulated cores, whereas (serialized) trace replay using gem5 scales with the simulated system size. Note that for the speedup results in the remainder of this section, we exclude the cost of trace generation due to two reasons. First, the trace generation overhead is amortized over multiple trace replay runs (Section IV-B). Second, the trace generation step is hardware-dependent, and can be sped up using a native machine with more cores and a faster secondary storage. Across all our experiments, the one-time cost for trace generation was 0.1×-2.5× the time for one trace replay run.

B. Accuracy and Correlation vs. Detailed gem5 Model

We first provide results that compare HETSIM simulations with those performed on the detailed gem5 model, for the same inputs and architectural configurations. We note that it is generally not possible to obtain 100% accurate estimation with HETSIM, due to limitations arising from a lack of support of pipelining, instrumentation at the LLVM IR level, among others. We list the major sources of inaccuracy in Section VII.

Due to space constraints, we only report results for two Transmuter configurations, (*i*) where the L1 and L2 are both configured as private caches, and (*ii*) where they are both configured in the shared mode. Specifically, we report the following metrics, where we note that deviation is calculated using the formula $(\text{metric}_{\text{HETSIM}} - \text{metric}_{\text{detailed}}) \,/\, \text{metric}_{\text{detailed}}$.

- Deviation in simulated time, *i.e.* the time to execute the workload on the target, as estimated by the simulator.
- Deviation in power consumption, *i.e.* the estimated average power for running the workload on the target.
- Deviation in L1 access count, *i.e.* the average number of accesses to each L1 cache bank in the Transmuter design.
- Wall clock speedup of HETSIM over the detailed gem5 model on the native machine.

We now analyze these metrics on Transmuter for the three workloads – GeMM, GeMV and SpMM, and compare them with those estimated by the MinorCPU-based detailed model.

GeMM. We evaluate blocked-GeMM on Transmuter while sweeping the matrix dimension and block size. Figure 7 shows our results. We achieved an average speedup of 5.0× using HETSIM. We note the following observations from our experiments. The L1 access count is underestimated by only 5.7% on average, showing that memory accesses are reproduced faithfully. The accuracy of performance and power estimates improve with the matrix dimension, varying from –2.6 to +16.2% for timing, and –21.9 to –5.3% for power. The deviations in estimates are visibly higher for the shared mode in comparison to the private mode, since the shared cache configuration involves much higher cache hits, and thus the timing is almost entirely dependent on the accuracy of simulating the compute operations.

GeMV. We evaluate blocked as well as unblocked versions of GeMV on Transmuter. From Figure 8, we observe that the blocked GeMV implementations have much lower runtime deviation with HETSIM (10.6%), in comparison to the unblocked version (30.0%). This is because the GPEs in the blocking version synchronize with LCPs after computing every "block size" elements, thus leading to self-correction of the errors introduced due to approximation of compute operations. Since GeMV is memory-bounded, the generated traces are majorly composed of memory operations that execute fairly accurately on HETSIM, translating to small deviations particularly for power (+2.0 to –5.7%). HETSIM executes, on average, 5.2× faster than the detailed model.

SpMM. We evaluate outer product based SpMM on Transmuter with uniformly-random synthetic sparse matrices. Figure 9 shows the results for 6 different matrices on the two Transmuter configurations. For the shared cache configuration, we note an increase in speedup as the matrix gets larger (fixed

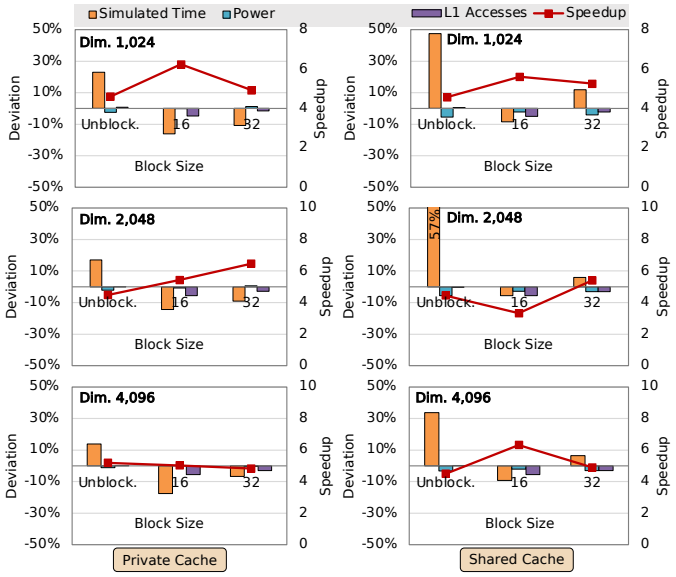


Fig. 8. Accuracy of performance, power and L1 cache access count estimations of HETSIM compared to the detailed gem5 model of 2×8 Transmuter, for GeMV with varying block sizes and two HETSIM configurations.

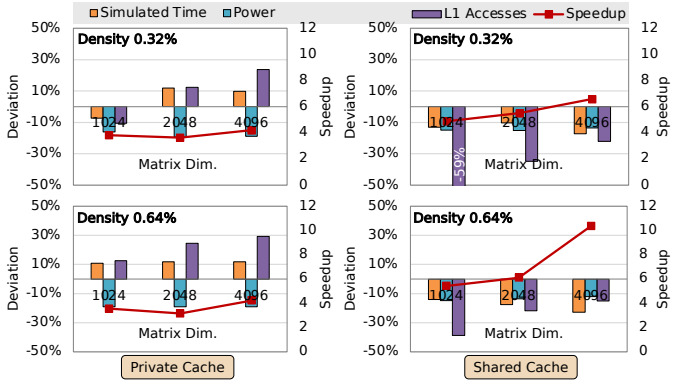


Fig. 9. Accuracy of performance, power and L1 cache access count estimations of HETSIM compared to the detailed gem5 model of 2×8 Transmuter, for SpMM with varying matrix dimension and density (uniform random), and two HETSIM configurations.

density). The timing deviation is slightly higher for SpMM, averaging 13.2%. This is however attributed to register spill/fill accesses that are omitted due to allow-listing only accesses to the SHARED_SPACE in HETSIM, as evidenced by the L1 access count deviation of up to –59%. Power consumed in the memory subsystem is underestimated due to the same reason.

C. Scaling with Problem Size and Configuration

We evaluate the performance predicted by the detailed gem5 model with HETSIM for the two Transmuter configurations. The detailed results for the three benchmarks are shown in Figures 10-12. We sweep the matrix and block size for GeMM and GeMV. For SpMM, we split the results for multiply and merge phases separately.

For GeMM, HETSIM made the same prediction about the better Transmuter configuration as the detailed model for 7 of the 8 reported experiments. For both GeMV and SpMM, HETSIM achieved 100% accuracy in predicting the faster Transmuter configuration, thus showcasing its efficacy for fast and accurate design space exploration.

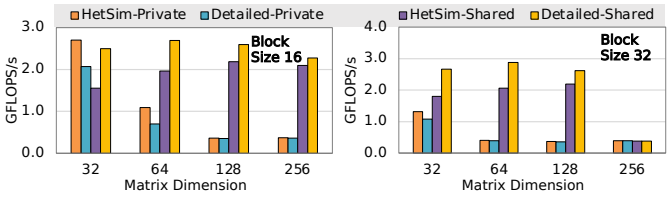


Fig. 10. Strong scaling performance comparison between different 2×8 Transmuter configurations on a blocked GeMM implementation, using HETSIM and the detailed model.

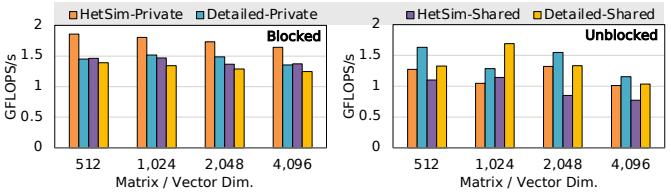


Fig. 11. Strong scaling performance comparison between different 2×8 Transmuter configurations on two GeMV implementations, using HETSIM and the detailed model.

D. Scaling with Cache Sizes

One typical use case of HETSIM is in design-space exploration of power-performance-area trade-offs by varying different parameters of the target hardware. We exemplify this through a Pareto frontier analysis of the L1 and L2 cache bank capacities in a 4×16 Transmuter. Figure 13 shows the power and performance with fixed input sizes for GeMM, GeMV and SpMM. The L1 and L2 cache bank sizes are varied from 1 kB to 64 kB. We observe that the greatest benefit of increasing cache capacity is observed for GeMV, followed by GeMV. SpMM does not see drastic performance differences with cache capacity because it is bottlenecked by cold misses.

E. Evaluation of SpMM Accelerator Chip

As with any architectural simulator, it is important to validate its estimations against real silicon. For this, we performed sensitivity analysis on the prior SpMM chip that uses outer product. Figure 14 (right) shows the variation of the chip's measured performance with memory bandwidth and number of fixed function PEs/merge cores. The average error in estimated performance of HETSIM compared to the chip is 32% for the multiply phase and 16% for the merge phase computation. We hypothesize that the higher error for multiply is due to the behavior of the custom off-chip memory interface, which we modeled approximately using the stock DDR model in gem5.

VII. DISCUSSION

HETSIM fundamentally trades off simulation accuracy for faster simulation speeds. We report here a list of known sources of error that directly impact the accuracy of HETSIM.

- Effect of not modeling pipelined execution within a PE, and instead assigning a fixed latency to each operation.
- Effect of frequent synchronization and busy-waiting (*e.g.* for mutexes and barriers), for which memory traffic is dynamic depending on factors such as contention.
- Differences due to trace generation on a system with a different ISA than that used by the target PE.
- Effect of instrumenting at the RISC-like LLVM-IR level.

Fig. 12. Strong scaling performance comparison between different 2×8 Transmuter configurations on an SpMM outer product implementation with uniform random matrices, using HETSIM and the detailed model.

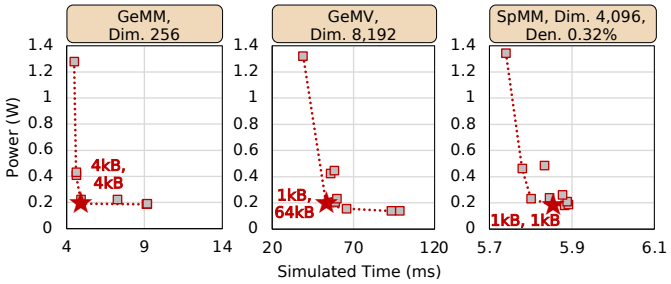


Fig. 13. Pareto analysis for GeMM, GeMV and SpMM on 4×16 Transmuter. The datapoints correspond to $\text{Cap}_{\text{L1}} \in \{1, 4, 16, 64\}$ kB, $\text{Cap}_{\text{L2}} \in \{1, 4, 16, 64\}$ kB, $\text{Cap}_{\text{L1}} \leq \text{Cap}_{\text{L2}}$. Star indicates the design with best energy-efficiency.

- Impact of over-filtering (deny-listing) primitives in the user specification file.
- For targets with programmable PEs, HETSIM ignores:
  - Effect of bandwidth sharing and performance/energy impact due to not accounting for I-cache misses.
  - Power impact of instruction fetch and decode in the pipeline, and SRAM accesses within I-caches.

HETSIM is under active development and is being used in a multi-University program as of this writing. We have released a version of HETSIM that works with the latest version of gem5, gem5-20 [52], as a public GitHub repository[1]. It contains an example target architecture that is intended to serve as a template for users to implement their own architectures. This initial release is bundled with the set of primitives in Table II, and can be extended with user-defined primitives. We encourage and appreciate user contributions to this repository.

We are also exploring orthogonal features, such as trace compression [5], that we will enable in future HETSIM releases. Another scope of our future work is to provide the option to select a trade-off between speed and accuracy that would better cater to the user's requirements.

VIII. CONCLUSION

We developed HETSIM as an end-to-end framework to speed up pre-silicon performance and power estimation of heterogeneous systems. HETSIM addresses the issue of simulating heterogeneous systems with *thousands* of cores within practical time and resource limitations. In contrast to existing frameworks, HETSIM introduces the notion of hardware primitives and implements them in a software emulation library that is exposed to the user application. Additionally, HETSIM supports complex cores and prefetching mechanisms by embedding crucial information, such as dependent memory addresses and program counter values, within its traces.

[1]GitHub repository: https://github.com/umich-cadre/HetSim-gem5

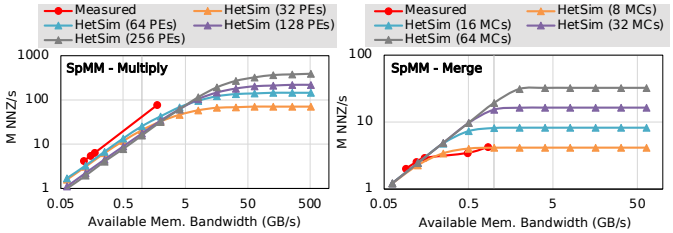


Fig. 14. Weak scaling characteristics with different memory bandwidth values for the SpMM chip running the two phases of outer product SpMM with matrix dimension = 100,000 and density = 0.008%.

In this work, we evaluated HETSIM on two target architectures and reported that HETSIM speeds up measured simulation times by 3.2×-10.4× over detailed gem5 models for the same targets. We also noted that HETSIM enables such fast design space exploration with a small impact in terms of accuracy of estimated performance and power. We observed deviations of 0.2%-57.0% and 0.0%-24.2% in terms of simulated time and power, respectively, for three different applications on the targets.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their feedback. The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

[1] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar *et al.*, "A simulator for large-scale parallel computer architectures," *IJDST*, 2010.

[2] J. Hestness, B. Grot, and S. W. Keckler, "Netrace: dependency-driven trace-based network-on-chip simulation," in *Third International Workshop on Network on Chip Architectures*, 2010, pp. 31–36.

[3] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, and N. When, "Exploring system performance using elastic traces: Fast, accurate and portable," *SAMOS*, 2017.

[4] A. Nocua, F. Bruguier, G. Sassatelli, and A. Gamatie, "ElasticSim-MATE: A fast and accurate gem5 trace-driven simulator for multicore systems," *ReCoSoC*, 2017.

[5] K. Sangaiah, M. Lui, R. Jagtap, S. Diestelhorst, S. Nilakantan *et al.*, "SynchroTrace: Synchronization-Aware architecture-Agnostic traces for lightweight multicore simulation of CMP and HPC workloads," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2018.

[6] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling."

[7] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 372–383.

[8] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[9] M. Badr and N. E. Jerger, "A high-level model for exploring multi-core architectures," *Parallel Computing*, vol. 80, pp. 23–35, 2018. [Online]. Available: https://github.com/mariobadr/rhythm

[10] S. Feng, S. Pal, Y. Yang, and R. G. Dreslinski, "Parallelism analysis of prominent desktop applications: An 18-year perspective," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 202–211.

[11] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu *et al.*, "Kilocore: A 32-nm 1000-processor computational array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 891–902, 2017.

[12] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[13] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *arXiv preprint arXiv:1811.06992*, 2018.

[14] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park *et al.*, "Sparse-tpu: Adapting systolic arrays for sparse matrices," in *International Conference on Supercomputing (ICS'20)*, 2020.

[15] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.

[16] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, "Alrescha: A lightweight reconfigurable sparse-computation accelerator," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 249–260.

[17] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng *et al.*, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[18] S. Pal, D.-h. Park, S. Feng, P. Gao, J. Tan *et al.*, "A 7.3 m output non-zeros/j sparse matrix-matrix multiplication accelerator using memory reconfiguration in 40 nm," in *2019 Symposium on VLSI Technology*. IEEE, 2019, pp. C150–C151.

[19] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 2011, pp. 365–376.

[20] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "Rispp: rotating instruction set processing platform," in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 791–796.

[21] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao *et al.*, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[22] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE transactions on computers*, vol. 49, no. 5, pp. 465–481, 2000.

[23] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, "Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 575–587.

[24] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *37th international symposium on microarchitecture (MICRO-37'04)*. IEEE, 2004, pp. 30–40.

[25] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[26] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The garp architecture and c compiler," *Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[27] "Parallel m5." [Online]. Available: http://www.m5sim.org/Parallel_M5

[28] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 183–192.

[29] L. Ceze, J. Tuck, and J. Torrellas, "Are we ready for high memory-level parallelism," in *4th Workshop on Memory Performance Issues*, 2006.

[30] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE transactions on computers*, vol. 44, no. 5, pp. 609–623, 1995.

[31] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 135–145.

[32] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *Ieee micro*, vol. 26, no. 4, pp. 52–60, 2006.

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[34] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead, "Synchrotrace: Synchronization-aware architecture-agnostic traces for light-weight multicore simulation," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 278–287.

[35] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[36] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 97–108.

[37] S. Rogers, J. Slycord, R. Raheja, and H. Tabkhi, "Scalable llvm-based accelerator modeling in gem5," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 18–21, 2019.

[38] 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 *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 380–387.

[39] H.-Y. Cheng, J. Zhan, J. Zhao, Y. Xie, J. Sampson, and M. J. Irwin, "Core vs. uncore: The heart of darkness," in *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[40] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[41] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1981, pp. 207–218.

[42] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[43] H. T. Kung and C. E. Leiserson, "Systolic arrays for (vlsi)." CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1978.

[44] W. J. Dally and H. Aoki, "Deadlock-free adaptive routing in multicomputer networks using virtual channels," *IEEE transactions on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 466–475, 1993.

[45] S. Pal, S. Feng, D. hyeon Park, S. Kim, A. Amarnath *et al.*, "Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration," in *IEEE 29th International Conference on Parallel Architectures and Compilation Techniques (PACT) 2020, Virtual, October 3-7, 2020*. IEEE, in press.

[46] A. Soorishetty, J. Zhou, S. Pal, D. Blaauw, H.-S. Kim *et al.*, "Accelerating linear algebra kernels on a massively parallel reconfigurable architecture," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 1558–1562.

[47] Y. Xiong, J. Zhou, S. Pal, D. Blaauw, H.-S. Kim *et al.*, "Accelerating deep neural network computation on a low power reconfigurable architecture," in *2020 IEEE International Symposiumon Circuits and Systems (ISCAS)*. IEEE, in press.

[48] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, and M. Bussmann, "Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library," in *International Conference on High Performance Computing*. Springer, 2017, pp. 496–514.

[49] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, "Optimizing cuda code by kernel fusion: application on blas," *The Journal of Supercomputing*, vol. 71, no. 10, pp. 3934–3957, 2015.

[50] D.-H. Park, S. Pal, S. Feng, P. Gao, J. Tan *et al.*, "A 7.3 m output non-zeros/j, 11.7 m output non-zeros/gb reconfigurable sparse matrix-matrix multiplication accelerator," *IEEE Journal of Solid-State Circuits*, 2020.

[51] J. E. Smith, "Decoupled access/execute computer architectures," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 112–119, 1982.

[52] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.