Towards a scalable functional simulator for the Adapteva Epiphany architecture

Ola Jeppsson and Sally A. McKee

Chalmers University of Technology, Gothenburg, SWEDEN
olaj@student.chalmers.se, mckee@chalmers.se

Abstract. For many decades, Moore's law allowed processor architects to consistently deliver higher-performing designs by increasing clock speeds and increasing Instruction-Level Parallelism. This approach also led to ever increasing power dissipation. The trend for the past decade has thus been to place multiple (simpler) cores on chip to enable coarser-grain parallelism and greater throughput, possibly at the expense of single-application performance.
Adapteva's Epiphany architecture carries this approach to an extreme: it implements simple RISC cores with 32K of explicitly managed memory, stripping away caches and speculative hardware and replacing a shared bus with a simple mesh. The result is a scalable, low-power architecture: the 64-core Epiphany-IV is estimated to deliver 70 (single precision) GFLOPS per watt. Adapting software to fully exploit this impressive design remains an open problem, though. The only available simulator prior to the work presented here models a single Epiphany core. We build on that to develop a scalable, parallel functional chip simulator. This tool is a work-in-progress: the next step will be to add timing models and DMA interfaces to faithfully simulate user applications at scale.

1 Introduction

For many decades, shrinking feature sizes have enabled more and more transistors on chip, which has allowed processor architects to consistently deliver higher-performing designs by raising clock speeds and adding more hardware to increase Instruction-Level Parallelism. Unfortunately, techniques like speculative and out-of-order execution not only increase performance — they also increase power consumption, which, in turn, increases heat dissipation.

For Intel, the industry leader for desktop computer microprocessors, the single-core era culminated in 2005. The company cancelled its Tejas and Jayhawk projects, which analysts attributed to heat problems [4]. Power and thermal considerations have thus become first-class design parameters. Instead of focusing on single-core performance, chip manufacturers have turned to multi-core designs to deliver greater parallel performance. Many (including Intel and Adapteva) predict 1000-core chips by the year 2020.

Before we began this project, there existed only a single-core Epiphany simulator: simulating an Epiphany chip with all cores running concurrently was not

possible. Without full-chip simulation, hardware and software design decisions must be made based on analysis and experience or prototyping. Having a scalable simulator makes it possible to explore richer hardware design spaces and makes it possible to develop and optimize scalable applications (even before the chips for which they are designed become available).[1]

2 Adapteva Epiphany

First, we briefly introduce the Epiphany architecture. Most information needed to implement the simulator can be found in the reference manual [2]. Other information has come from the community forums and, in a few cases, testing things on hardware.

The Epiphany architecture is a many-core processor design consisting of "tiny" RISC cores connected with a 2D mesh on-chip network. Both the cores and the NoC are designed for scalability and energy efficiency, and therefore many things one would expect in modern multicore processors have been stripped away. For instance, there is no inter-core bus, no cache (and thus no coherence protocol), and no speculation (not even branch prediction). The result is an energy-efficient architecture that can achieve up to 70 GFLOPS/Watt [2] and scale to thousands of cores.

2.1 RISC Cores

Each core is a simple RISC with dual-issue pipelines, one single-precision FPU, and two integer ALUs. The ISA includes about 40 instructions (depending on how you count). A core can execute two floating-point operations (multiply-accumulate) and one load per clock cycle. The register file has 64 general-purpose registers, and all registers are memory mapped. Each core has two event timers, and the interrupt controller supports nested interrupts. DMA units per core support two parallel transactions.

[1] The Adapteva community has over 5,000 registered members, and over 10,000 evaluation boards have been sold. Our simulator thus has the potential for high impact.

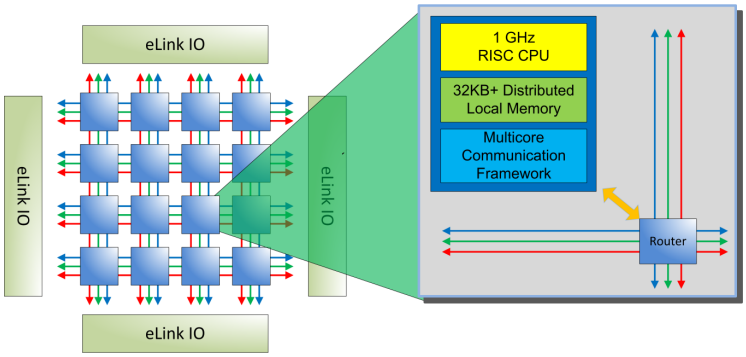

Fig. 1: Epiphany Architecture [2]

2.2 Network-on-Chip

Figure 1 shows the 2D mesh layout of the NoC. The Epiphany implements separate networks for different types of traffic: one for reads (the *rMesh*), one for on-chip writes (the *cMesh*), and one for off-chip writes (the *xMesh*). We collectively refer to these as the *eMesh*. Messages are first routed east-west, then north-south. Four *eLink* routers connect the cores on their north, south, east, and west edges. Assuming a clock frequency of 1GHz, total off-chip bandwidth is 6.4 GB second, and total on-chip network bandwidth is 64 GB per second at every core router.

2.3 Memory Model

The architecture has a shared, globally addressable 32-bit (4GB) memory address space. An address is logically divided into the *coreID* (upper 12 bits) and offset (lower 20 bits). Thus, every core can have up to 1MB of globally addressable memory. The upper six bits of the coreID determine the core's row; the lower six determine its column. An address with coreID zero aliases to the local core's memory region. The architecture thus supports up to 4095 cores.

The architecture supports *TESTSET* (for synchronization), *LOAD*, and *STORE* memory operations. The upper 64KB of each core's memory region consists of memory-mapped registers. The configurations on the market (Epiphany-III and Epiphany-IV) also map a portion of the address space to 32MB of external RAM.

All local memory accesses have strong memory ordering, i.e., they take effect in the same order as issued. Memory accesses routed through any of the three NoCs have a weaker memory ordering. The router arbitration and dispatch rules are deterministic, but the programmer is not allowed to make assumptions regarding synchronization, since there is no way to know the global "system state". Section 4.2 of the reference manual [2] lists the only guarantees on which the programmer can depend:

1. All local memory accesses have a strong memory ordering, i.e., they take effect in the same order as they were issued;
2. All memory requests that enter the NoC obey the following:
   (a) *LOAD* operations complete before the returned data is used by a subsequent instruction,
   (b) *LOAD* operations using data previously written use the updated values, and
   (c) *STORE* operations eventually propagate to their ultimate destination.

3 The GNU Debugger

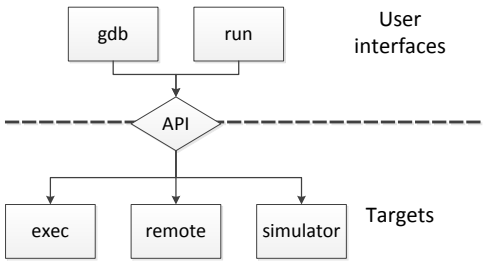

Fig. 2: `gdb` Overview

The simulator uses the common simulator framework for `gdb` (the GNU debugger), which is widely used and serves as the defacto standard debugger in the open-source community. Written by Richard Stallman in the 1980s, it was maintained by Cygnus Solutions throughout the 1990s until they merged with Red Hat in 1999. During this time `gdb` gained most of its target support, and many `gdb`-based simulators were written. Like the Adapteva core simulator on which we base our work, most of these model embedded systems.

`gdb` is divided into three main subsystems: user interface, target control interface, and symbol handling [6]. Simulators are most concerned with the user interface and target control interface. Compiling `gdb` with the Epiphany simulator target creates two binaries, `epiphany-elf-gdb` and `epiphany-elf-run`. `epiphany-elf-gdb` is linked with the target simulator and presents the standard `gdb` user interface. `epiphany-elf-run` is a stand-alone tool that connects to the simulator target and runs a binary provided as a command-line argument.

3.1 Simulator Framework

The GNU toolchain (`binutils`, `gcc`, `gdb`) has been ported to many architectures, and since writing a simulator in the process makes it easier to test generated

code, `gdb` has acquired several simulator targets. The process generally follows these steps:

- define the CPU components (register file, program counter, pipeline), instruction set binary format, and instruction semantics in a CPU definition file;
- write architecture-specific device models;
- write needed support code for a main-loop generator script; and
- write simulator interface code.

The CPU definition file is written in an embedded Scheme-based domain specific language. That definition is fed through CGEN [3] (CPU tools GENerator) to create C files for instruction decoding and execution within the simulator framework. Since code for the simulator interface and main loop tends to be similar across architectures, an existing simulator target can often be used as a base. For example, parts of the Epiphany implementation originate from an Mitsubishi M32R port.

4 Implementation

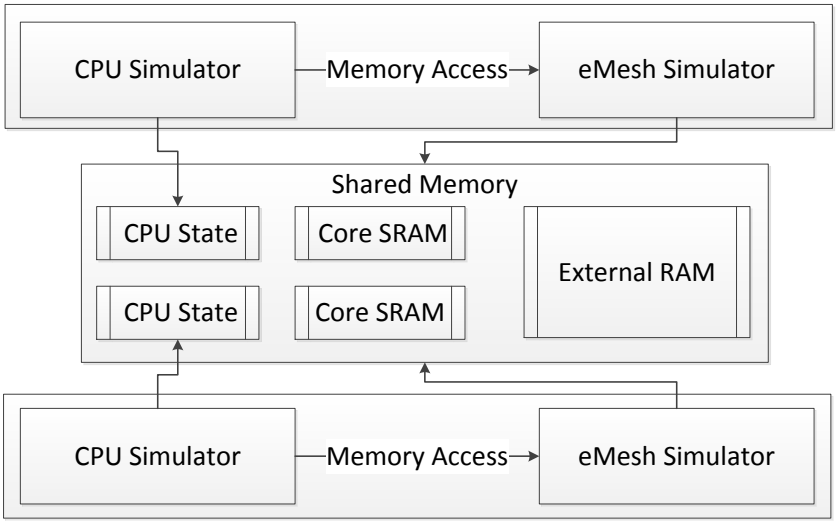

Fig. 3: Simulator Overview

We next discuss details of our simulator implementation. Figure 3 shows how we extend the single-core simulator to model a many-core system. Our design is process-based: for every core in the simulated system we launch an

`epiphany-elf-run` process. When the Epiphany simulator target is initialized, it also initializes the mesh simulator, which connects to a shared memory file. The mesh network simulator uses POSIX shared memory to connect relevant portions of each core simulator via a unified address space, and all memory requests are routed through this mesh simulator. The register file resides in the `cpu_state` structure. Since the mesh simulator needs to access remote CPU state for some operations, we also store that state in the shared address space.

4.1 Single-Core Simulator Integration

The core simulator upon which we build was written by Embecosm Ltd [1]. on behalf of Adapteva. Most instruction semantics had already been defined, but due to the design of the `gdb` simulator framework, the simulator lacked support for self-modifying code in the modeled system. Due to the small local memories (32KB) in the Epiphany cores, executing code must be able to load instructions dynamically (like software overlays). Enabling self-modifying code required that we modify software mechanisms intended to speed simulation. For instance, a semantics cache maintains decoded target machine instructions, and in the original simulator code, writes to addresses in the semantics cache would update memory but not invalidate the instructions in the cache.

We map the entire simulated 32-bit address space to a "shim" device that forwards all memory requests to the eMesh network simulator. Recall that the CPU state of all cores resides in the shared address space, where the eMesh simulator can access it easily.

Algorithm 1 shows pseudo code for the simulator main loop. The highlighted lines are from the original single-core main loop. Lines 1-6 and 9 are inserted by the main-loop generator script. The instruction set has an *IDLE* function that puts the core in a low-power state and disables the program sequencer. We implement something similar in software: in line 8 we check whether the core is active, and if not, we sleep until we receive a wakeup event.

In line 13 we check whether another core has a pending write request to a special core register. Writes to special core registers are serialized on the target core because they might alter internal core state. In line 10 and 18 we handle out-of-band events. Such events might affect program flow and are triggered by writes to special core registers, e.g., by interrupts or reset signals.

In line 19 we ensure that only instructions inside the core's local memory region can ever reside in the semantics cache. Without this constraint we would need to do an invalidate call to all cores' semantics caches on all writes. In line 21 we check whether the external write flag is set, and if so, we flush the entire semantics cache. This flag is always set on a remote core when there is a write to that core's memory.

4.2 eMesh Simulator

As shown in Figure 3, the eMesh simulator creates a shared address space accessible to all simulated cores. This is accomplished via the POSIX shared memory

Algorithm 1: Main Loop (simplified for illustration)

```
   Highlighted lines are the original main loop
 1 while True do
 2     sc ← scache.lookup(PC);
 3     if sc = ∅ then
 4         insn ← fetch_from_memory(PC);
 5         sc ← decode(insn);
 6         scache.insert(PC, sc);
 7     old_PC ← PC;
 8     if core is in active state then
 9         PC ← execute(sc);
10         PC ← handle_out_of_band_events(PC);
11     else
12         wait_for_wakeup_event();
13     if ext_scr_write_slot.reg ≠ -1 then
14         reg_write(ext_scr_write_slot.reg,
15                   ext_scr_write_slot.value);
16         ext_scr_write_slot.reg ← -1;
17         signal_scr_write_slot_empty();
18         PC ← handle_out_of_band_events(PC);
19     if old_PC ∉ local memory region then
20         scache.invalidate(old_PC);
21     if external_mem_write_flag then
22         scache.flush();
23         external_mem_write_flag ← False;
```

API. We use POSIX threads (pthreads) for inter-process communication.

The eMesh simulator provides an API for memory transactions (*LOAD*, *STORE*, *TESTSET*) and functions to connect and disconnect to the shared address space. We also provide a client API so that other components can access the Epiphany address space (e.g., to model the external host or to instrument a simulated application).

Every memory request must be translated. The translator maps an Epiphany address to its corresponding location in the simulator address space. It also determines to which type of memory (core SRAM, external DRAM, memory-mapped registers [general-purpose or special-core], or invalid) the address corresponds.

How the request is serviced depends on the memory type. Accesses to core SRAM and external DRAM are implemented as native load and store operations (the target core need not be invoked). Memory-mapped registers are a little trickier. All writes to such registers are serialized on the target core, which is accomplished with one write slot, a mutex, and a condition variable. Reads to special core registers are implemented via normal load instructions. Since reads to memory-mapped, reads to general-purpose registers are only allowed when the target core is inactive; we check core status before allowing the request.

We created a backend for the Epiphany hardware abstraction layer (*e-hal*). Using the client API lets us compile Parallella host applications natively for x86_64 without code modification[2]. We experimented with cross-compiling programs (from the `epiphany-examples` repository on the Adapteva github account) with generally good results. Obviously, programs that use implicit synchronization might not work, and programs that use core functionalities not yet supported will not work.

We have also extended the mesh simulator with networking support implemented in MPI [7]. We use MPI's RMA (remote memory access) API to implement normal memory accesses (core SRAM and external RAM). We implement all register accesses with normal message passing and a helper thread on the remote side. We implement *TESTSET* with `MPI_compare_and_swap()`, which is only available in MPI-3.0 [5]. Since we use both threads and MPI-3.0 functionalities, we require a fairly recent MPI implementation compiled with `MPI_THREADS_MULTIPLE` support.

4.3 `epiphany-elf-sim`

As noted, the simulator is process-based, i.e., one simulated core maps to one system process. When we began development, we started these processes by hand (which is cumbersome and does not scale). We therefore created a command-line tool that makes it easy to launch simulations. Mesh properties and the program(s) that should be loaded onto the cores are given as command-line arguments, and the tool spawns and manages the core simulator processes.

[2] All data structures must have the same memory layout in both the 32-bit and 64-bit versions.

5 Limitations and Future Work

The current simulator is purely functional, and thus it lacks a timing model. Adding such a model will enable more accurate performance analysis.

Before a simulation starts, all core simulators synchronize on a barrier, but thereafter the cores can "drift apart". Even though the simulator remains functionally correct, making implicit timing assumptions can still render programs faulty. The core simulators thus require a sense of global time.

The two most notable missing CPU features are DMA and event timers. We believe that DMA functionality can be implemented adequately with the current eMesh simulator design. For the event timers, some of the sources are harder to implement than others. Some require a more accurate pipeline timing model, which CGEN supports to some extent. Other events are generated when packets flow through the NoC-router connected to the CPU.

We would like to add support for more advanced mesh features like multicast and user-defined routing. Message passing presents one obvious solution for more accurate routing. We could make MPI a hard dependency and spawn an extra router thread in each simulator process. Alternatively, we could use event queues (like `MPI_Send()` but without MPI), or we could model the mesh as a directed graph and let the initiating core route the request all the way to the target core, using locks on the edges for sequencing messages. This should trigger fewer context switches.

6 Preliminary Results

The simulator currently supports most features not marked as LABS in the reference manual (i.e., untested or broken functionalities), with the exceptions of DMA and event timers. The simulator executes millions of instructions per second (per core) and scales up to 4095 cores running concurrently on a single computer. With respect to the networking backend, we have run tests with up to 1024 simulated cores spread over up to 48 nodes in an HPC environment. In larger single-node simulations the memory footprint averages under 3MB per simulated core. Note that the simulator design requires that writes from non-local (external) cores flush the entire semantics cache (see algorithm 1, line 21-23) rather than just invalidating the affected region, which may hamper performance.

6.1 Matrix multiplication

The parallel matrix multiplication application from `epiphany-examples` uses many CPU functions and performs all interesting work on the Epiphany chip, and thus we choose it for initial studies of simulator behavior. For simplicity, we move all host code to the first core process. We implement our own data transfer functions, since the simulator does not yet support DMA. Our port revealed a race condition in the `e-lib` barrier implementation, which we attempted to fix (see the discussion of the 1024-core simulation, below).

For the tests, we allocated 32 nodes on an HPC cluster. Nodes contain two Intel Xeon processors with 16 physical cores (eight per socket) connected by a Mellanox Infiniband FDR. Hyper-threading is disabled. Tables 1-4 present results for 16, 64, 256, and 1024 cores per computation, respectively.

Things to consider:

1. Using more nodes creates more network traffic (versus direct accesses to memory within a node) for handling simulated memory accesses. This is orders of magnitude slower, even with FDR Infiniband.
2. More nodes means more physical cores. If all cores only accessed their local memory, the ideal number of nodes would be where there was a one-to-one mapping between physical cores and simulated cores.
3. The barrier implementation does busy-waiting, and thus waiting for another core increases total instructions executed. This also shows a clear limitation of the simulator. Because there is no global time or rate limiting, the number of executed instructions could differ significantly between the simulator and real hardware.
4. The nodes are allocated with exclusive access, but the network is shared between all users of the cluster. This means that there might be network contention or other system noise that could qualitatively affect results. Another error factor is the network placement of the nodes for a requested allocation, which is also beyond our control.

For matmul-16 and matmul-64, results are understandable: the number of instructions executed per (simulated) core per second increases until we reach a 2:1 mapping between physical and simulated cores. The reason that 2:1 outperforms 1:1 is likely because all simulator processes running on a node can be pinned to (physical) cores on the same socket, which means more shared caches and less cache coherence communication between sockets.

Execution time increases for a modest number of simulated cores when we go from one to two nodes due to network communication. We get a nice speedup in execution time for matmul-64.

For matmul-256 two results stand out. The jump in execution time from one to two nodes is much higher compared to matmul-16 and matmul-64. This data point might be due to a system/network anomaly: unfortunately, we only ran this test once (to date), so we cannot yet explain the observed behavior.[3] For matmul-256 running on 32 nodes, execution time jumps from 44 seconds on 16 nodes to 521 seconds on 32 nodes. We could expect execution time to increase a bit, since there is a 1:1 mapping between physical and simulated cores on 16 nodes, and we get more network traffic with 32 nodes. We repeated the test a few times (on the same allocation) with similar results. This behavior, too, requires further study.

When we tried to scale up to 1024 simulated cores, the program could not run to completion. Attaching the debugger revealed that all simulated cores

[3] In truth, our allocation of CPU hours expired before we could repeat all our experiments. We are in the process of repeating all experiments on multiple clusters, but results will not be available before this paper goes to press.

were stuck waiting on the same barrier. It is likely that we hit the `e-lib` race condition and that our fix proved insufficient. As a limit study, we removed all barrier synchronization. This means that program output is incorrect, and the results for matmul-1024 are not directly comparable to the other runs. Since all synchronization is removed, we expect execution time to be lower than it would have been in a correct implementation. But the program still exhibits a similar memory access pattern, so it is fair to assume that the instruction rate tells something about performance even with synchronization back in place. From the previous results, we would expect running on 128 nodes to yield the lowest execution time and peak instruction rate; running on a larger allocation is part of future work.

Nodes	Execution time	Total insns	Min insns	Max insns	Avg. insns/core/s	Avg. insns/s
1	32.0 s	2.28E+09	1.40E+08	1.44E+08	4.45E+06	7.12E+07
2	37.9 s	4.22E+09	2.07E+08	3.29E+08	6.96E+06	1.11E+08
4	53.2 s	5.13E+09	2.49E+08	4.68E+08	6.03E+06	9.65E+07
8	65.8 s	5.51E+09	2.42E+08	5.55E+08	5.24E+06	8.38E+07

Table 1: matmul-16

Nodes	Execution time	Total insns	Min insns	Max insns	Avg. insns/core/s	Avg. insns/s
1	81.6 s	6.01E+09	9.22E+07	9.52E+07	1.15E+06	7.37E+07
2	88.6 s	1.30E+10	1.26E+08	2.42E+08	2.30E+06	1.47E+08
4	46.5 s	1.29E+10	1.30E+08	2.52E+08	4.34E+06	2.78E+08
8	29.2 s	1.42E+10	1.33E+08	3.05E+08	7.60E+06	4.86E+08
16	29.9 s	1.46E+10	1.29E+08	3.16E+08	7.63E+06	4.88E+08
32	35.1 s	1.65E+10	1.52E+08	3.70E+08	7.35E+06	4.71E+08

Table 2: matmul-64

Nodes	Execution time	Total insns	Min insns	Max insns	Avg. insns/core/s	Avg. insns/s
1	344.3 s	2.10E+10	7.45E+07	8.51E+07	2.38E+05	6.10E+07
2	1007.3 s	1.52E+11	1.13E+08	9.18E+08	5.89E+05	1.51E+08
4	323.8 s	1.08E+11	1.51E+08	5.08E+08	1.31E+06	3.35E+08
8	103.0 s	6.98E+10	9.17E+07	3.53E+08	2.65E+06	6.77E+08
16	44.4 s	6.03E+10	1.36E+08	2.49E+08	5.31E+06	1.36E+09
32	520.9 s	1.58E+12	8.39E+07	6.32E+09	1.19E+07	3.04E+09

Table 3: matmul-256

Nodes	Execution time	Total insns	Min insns	Max insns	Avg. insns/core/s	Avg. insns/s
16	262.8 s	1.50E+11	3.12E+07	1.47E+08	5.59E+05	5.73E+08
32	135.5 s	1.50E+11	3.12E+07	1.47E+08	1.08E+06	1.11E+09

Table 4: matmul-1024*

7 Conclusions

We have modified a single-core Epiphany simulator, integrating it with our own mesh simulator that models the Epiphany network-on-chip. This enables full chip simulations modeling large numbers of cores. Although most of the necessary basic functionality is in place, the tool is still a work-in-progress, and we welcome others who would like to contribute to its further development. Source code is available from `https://github.com/adapteva/epiphany-binutils-gdb/tree/epiphany-gdb-7.6-multicore-sim` (the `epiphany-gdb-7.6-multicore-sim` branch). It is included as an experimental feature in the Epiphany SDK as of version 2014.11.

References

1. Embecosm Ltd. URL: `http://www.embecosm.com`.
2. Adapteva Inc. *Epiphany Architecture Reference*, Mar. 2014. URL: `http://www.adapteva.com/docs/epiphany_arch_ref.pdf`.
3. D. Evans and et al. CGEN: CPU tools GENerator. URL: `https://sourceware.org/cgen/`.
4. M. Flynn and P. Hung. Microprocessor design issues: Thoughts on the road ahead. *Micro, IEEE*, 25(3):16–31, May 2005. `doi:10.1109/MM.2005.56`.
5. M. P. I. Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.
6. J. Gilmore and S. Shebs. GDB Internals. Technical report, 2013.
7. W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.