Simulating a Manycore Processor:

A Scalable Functional Model for the Adapteva Epiphany Architecture

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

Sally A. McKee
Chalmers University of Technology
Gothenburg, SWEDEN
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. How to adapt software to fully exploit this impressive design remains an open problem. Hardware/software codesign would have developed a simulator that application programmers could use prior to the availability of the actual chips, but the only simulator available prior to the work presented here models a single Epiphany core. We build on that to develop a scalable, parallel functional chip simulator.

1. INTRODUCTION

For many decades, shrinking feature sizes have enabled more and more transistors on chip, which allowed processor architects to consistently deliver higher-performing processors 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 instance, the single-core era culminated in 2005 for Intel, the industry leader for desktop computer microprocessors. The company cancelled its Tejas and Jayhawk projects, which analysts attributed to heat problems [4]. Power and thermal considerations have now become first-class design parameters. Instead of focusing on single-core performance, chip manufacturers have turned to multicore designs to deliver greater parallel performance in their processor chips. 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 by prototyping. Having a scalable simulator makes it possible to explore richer hardware design spaces (and would be welcomed by the Adapteva designers) 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

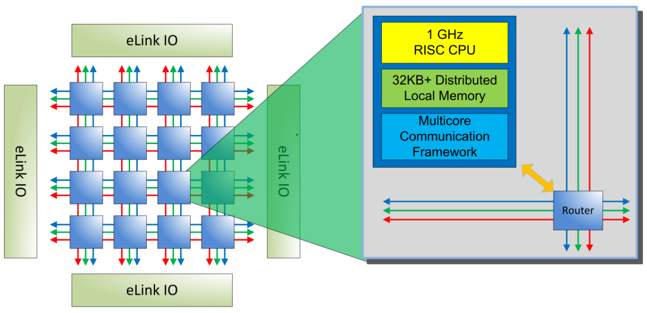
First, we briefly introduce the Epiphany architecture. Most information needed to implement the simulator can be found in the reference manual [2]. Other information has come from the community forums and, in a few cases, testing things on hardware.

The Epiphany architecture is a many-core processor design consisting of "tiny" RISC cores connected in a 2D mesh on-chip network. Both the cores and the NoC have been designed for scalability and energy efficiency, and therefore many things one would expect in modern multicore processors have been stripped away. For instance, there are no inter-core buses, no caches (and no cache-coherency protocol), and no speculation (not even branch prediction). The result is a very energy-efficient architecture that can achieve up to 70 GFLOPS/Watt [2] and scale to 1000s of cores.

2.1 RISC Cores

The cores are simple RISCs 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 (e.g., multiply-accumulate) and one load operation per clock cycle. The cores have no caches. The register file has 64 general-purpose registers. 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.


eLink IO
eLink IO
eLink IO
eLink IO
1 GHz RISC CPU
32KB+ Distributed Local Memory
Multicore Communication Framework
Router
Figure 1: Epiphany Architecture [2]

All local memory accesses have a strong memory ordering, i.e., they take effect in the same order as they were issued.
All memory requests that enter the NoC obey the following:
Load operations complete before the returned data is used by a subsequent instruction
Load operations using data previously written use the updated values
Store operations eventually propagate to their ultimate destination

Table 1: Memory Ordering Guarantees

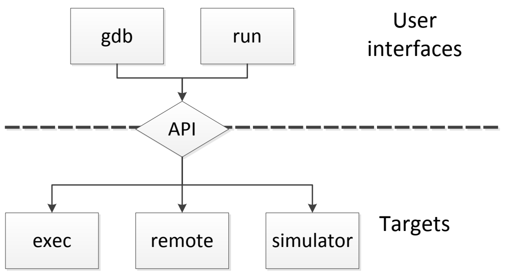


Figure 2: `gdb` Overview

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. Total off-chip bandwidth is 6.4 GB/s, and total on-chip network bandwidth is 64 GB/s at every core router, assuming a clock frequency of 1GHz.

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 address a dedicated 1 MB memory region. The upper six bits of the coreID determine the core's row; the lower six determine its column. An address with coreID is set to 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* operations. The upper 16th of each core's memory region holds 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 a strong memory ordering, i.e., they take effect in the same order as they were issued. Memory accesses routed through one 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 (summarized in Table 1).

3. THE GNU DEBUGGER

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 simulator on which we base our work, most of these are for 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 a simulator target creates two binaries, `epiphany-elf-gdb` and `epiphany-elf-run`. `epiphany-elf-gdb` is linked with the target simulator (in our case the Epiphany 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 over the years, and since writing a simulator in the process makes it easier to test generated code, `gdb` has acquired several simulator targets. The process generally includes 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 devices;
- 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

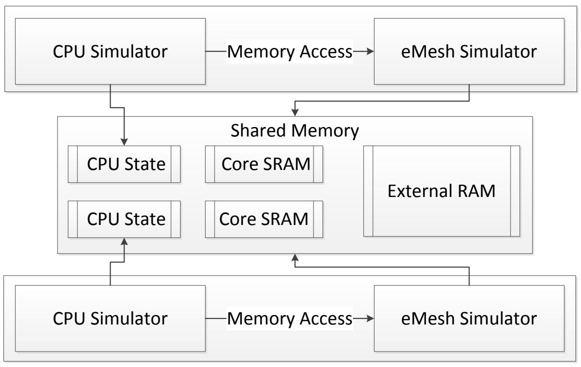


Figure 3: Simulator Overview

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 a 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. All memory requests are routed through the mesh simulator. The register file resides in the `cpu_state` structure, but since the mesh simulator for some operations needs to access remote CPU state, we also store that 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 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 the original single-core main loop. Lines 1-6 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 7 we check whether the core is in the active state, and if not, we sleep until we get a wakeup event.

In line 12 we ensure that only instructions inside the core’s local memory region can ever reside in the semantics cache.

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 $sc = \emptyset$ then
4          insn ← fetch_from_memory($PC$);
5          sc ← decode(insn);
6          scache.insert($PC$, sc);
7      $old\_PC \leftarrow PC$;
8      if core is in active state then
9          $PC$ ← execute(sc);
10     else
11         wait_for_wakeup_event();
12     if $old\_PC \notin$ local memory region then
13         scache.invalidate($old\_PC$);
14     if external_mem_write_flag then
15         scache.flush();
16         external_mem_write_flag ← False;
17     if ext_scr_write_slot.reg $\neq$ -1 then
18         reg_write(ext_scr_write_slot.reg,
19                   ext_scr_write_slot.value);
20         ext_scr_write_slot.reg ← -1;
21         signal_scr_write_slot_empty();
22     $PC$ ← handle_out_of_band_events($PC$);
```

Without this constraint we would need to do an invalidate call to all cores’ semantics caches on all writes. In line 14 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.

In line 17 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. Finally, in line 23 we handle out-of-band events that might have occurred. Such events might affect program flow and are usually triggered by writes to special core registers, e.g., by interrupts or reset signals.

4.2 eMesh Simulator

As shown in Figure 3, the eMesh simulator creates a shared address space accessible by all simulated cores. This is accomplished via the POSIX shared memory API. We use POSIX threads (pthreads) for interprocess communication.

The eMesh simulator provides an API for all types of 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 address translator maps an address in the Epiphany address space 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.

After the address is translated, the request is serviced. How this is done depends on the type of memory. Memory accesses to core SRAM and external DRAM are implemented as native loads and store operations; the target core need not be invoked.

Memory mapped registers are a little trickier. All writes to memory-mapped registers are serialized on the target core. This 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 from memory-mapped general-purpose registers are only allowed when the target core is inactive, we must check this before allowing the request.

We created a backend for e-hal (the Epiphany hardware abstraction layer) that uses the client API. This lets us compile Parallella host applications natively for x86_64 without code modification[2]. We experimented with cross compiling some example programs (from the `epiphany-examples` directory on the Adaptiva github account) with generally good results. Obviously, programs that use implicit synchronization or use core functionality not yet supported do not work.

We have also extended the mesh simulator with networking support. This is implemented in MPI [7] and is a compile-time option. We use MPI's RMA (remote memory access) API to implement normal memory access (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 with MPI-3.0 [5]. Since we use both threads and MPI-3.0 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.

5. RESULTS

The simulator currently supports most features not marked as LABS (i.e., untested or broken functionalities) in the reference manual, with the exception of DMA and event timers. The simulator can execute millions of instructions per second and scales up 4095 cores running on a single computer. With the networking backend, we have run tests with up to 768 simulated cores spread over 48 nodes in an HPC environment. In larger single-node simulations the memory footprint averages under 3MB per simulated core. The biggest design problem from a performance perspective is that writes from non-local (external) cores result in flushing the entire semantics cache (see algorithm 1, line 14) instead of just invalidating the affected region.

6. FUTURE WORK

The two most notable missing CPU features are DMA and event timers. We believe that DMA 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 requires a more accurate instruction timing model. Some of the possible event sources are emitted when packets flow through the NoC-router connected to the CPU.

We would like to add support for more advanced mesh features like multi-cast and user-defined routing rules. One obvious solution for more accurate routing is message passing. We could make MPI a hard dependency and spawn an extra router thread in every simulator process. 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 result in fewer context switches.

Before a simulation starts, all cores synchronize on a barrier, but after that the cores are not rate-limited and can "drift apart". Even though the simulator remains functionally correct, making implicit timing assumptions can still render programs faulty. The core simulators need to have a sense of global time.

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

7. CONCLUSIONS

We have modified a single-core Epiphany simulator and integrated it with our own mesh simulator to model the Epiphany network-on-chip. This has enabled us to do 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 http://github.com/olajep/epiphany-gdb and will likely soon be included in the Epiphany Parallella SDK.

8. 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.

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