Heracles: A Tool for Fast RTL-Based Design Space Exploration of Multicore Processors

Heracles: A Tool for Fast RTL-Based Design Space Exploration of Multicore Processors

Michel A. Kinsy Srinivas Devadas
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
mkinsy, devadas@mit.edu

Michael Pellauer
Intel Corporation
VSSAD Group
michael.i.pellauer@intel.com

ABSTRACT

This paper presents *Heracles*, an open-source, functional, parameterized, synthesizable multicore system toolkit. Such a multi/many-core design platform is a powerful and versatile research and teaching tool for architectural exploration and hardware-software co-design. The *Heracles* toolkit comprises the soft hardware (HDL) modules, application compiler, and graphical user interface. It is designed with a high degree of modularity to support fast exploration of future multicore processors of different topologies, routing schemes, processing elements (cores), and memory system organizations. It is a component-based framework with parameterized interfaces and strong emphasis on module reusability. The compiler toolchain is used to map C or C++ based applications onto the processing units. The GUI allows the user to quickly configure and launch a system instance for easy factorial development and evaluation. Hardware modules are implemented in synthesizable Verilog and are FPGA platform independent. The *Heracles* tool is freely available under the open-source MIT license at:
http://projects.csail.mit.edu/heracles.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Processor Architecture - Single-instruction-stream, multiple-data-stream processors (SIMD); B.5.1 [**Hardware**]: Register-Transfer-Level Implementation- Design.

General Terms

Tool, Design, Experimentation, Performance

Keywords

Multicore Architecture Design, RTL-Based Design, FPGA, Shared Memory, Distributed Shared Memory, Network-on-Chip, RISC, MIPS, Hardware Migration, Hardware multithreading, Virtual Channel, Wormhole Router, NoC Routing Algorithm.

1. INTRODUCTION

The ability to integrate various computation components such as processing cores, memories, custom hardware units, and complex network-on-chip (NoC) communication protocols onto a single chip has significantly enlarged the design space in multi/many-core systems. The design of these systems requires tuning of a large number of parameters in order to find the most suitable hardware configuration, in terms of performance, area, and energy consumption, for a target application domain. This increasing complexity makes the need for efficient and accurate design tools more acute.

There are two main approaches currently used in the design space exploration of multi/many-core systems. One approach consists of building software routines for the different system components and simulating them to analyze system behavior. Software simulation has many advantages: i) large programming tool support; ii) internal states of all system modules can be easily accessed and altered; iii) compilation/re-compilation is fast; and iv) less constraining in terms of number of components (e.g., number of cores) to simulate. Some of the most stable and widely used software simulators are Simics [14]–a commercially available full-system simulator–GEMS [21], Hornet [12], and Graphite [15]. However, software simulation of many-core architectures with cycle- and bit-level accuracy is time-prohibitive, and many of these systems have to trade off evaluation accuracy for execution speed. Although such a tradeoff is fair and even desirable in the early phase of the design exploration, making final micro-architecture decisions based on these software models over truncated applications or application traces leads to inaccurate or misleading system characterization.

The second approach used, often preceded by software simulation, is register-transfer level (RTL) simulation or emulation. This level of accuracy considerably reduces system behavior mis-characterization and helps avoid late discovery of system performance problems. The primary disadvantage of RTL simulation/emulation is that as the design size increases so does the simulation time. However, this problem can be circumvented by adopting synthesizable RTL and using hardware-assisted accelerators–field programmable gate arrays (FPGAs)–to speed up system execution. Although FPGA resources constrain the size of design one can implement, recent advances in FPGA-based design methodologies have shown that such constraints can be overcome. HAsim [18], for example, has shown using its time multiplexing technique how one can model a shared-memory multicore system including detailed core pipelines, cache hierarchy, and on-chip network, on a single FPGA. RAMP Gold [20] is able to simulate a 64-core shared-memory target machine capable of booting real operating systems running on a single Xilinx Virtex-5 FPGA board. Fleming et al [7] propose a mechanism by which complex designs can be efficiently and automatically partitioned among multiple FPGAs.

RTL design exploration for multi/many-core systems nonetheless remain unattractive to most researchers because it is still a time-consuming endeavor to build such large designs from the ground up and ensure correctness at all levels. Furthermore, researchers are generally interested in one key system area, such as processing core and/or memory organization, network interface, interconnect network, or operating system and/or application mapping. Therefore, we believe that if there is a platform-independent design framework, more specifically, a general hardware toolkit, which allows designers to compose their systems and modify them at will and with very little effort or knowledge of other parts of the system, the speed versus accuracy dilemma in design space exploration of many-core systems can be further mitigated.

To that end we present *Heracles*, a functional, modular, synthesizable, parameterized multicore system toolkit. It is a powerful and versatile research and teaching tool for architectural exploration and hardware-software co-design. Without loss in timing accuracy and logic, complete systems can be constructed, simulated and/or synthesized onto FPGA, with minimal effort. The initial framework is presented in [10]. *Heracles* is designed with a high degree of modularity to support fast exploration of future multicore processors–different topologies, routing schemes, processing elements or cores, and memory system organizations by using a library of components, and reusing user-defined hardware blocks between different system configurations or projects. It has a compiler toolchain for mapping applications written in C or C++ onto the core units. The graphical user interface (GUI) allows the user to quickly configure and launch a system instance for easily-factored development and evaluation. Hardware modules are implemented in synthesizable Verilog and are FPGA platform independent.

2. RELATED WORK

In [6] Del Valle *et al* present an FPGA-based emulation framework for multiprocessor system-on-chip (MPSoC) architectures. LEON3, a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture, has been used in implementing multiprocessor systems on FPGAs. Andersson *et al* [1], for example, use the LEON4FT microprocessor to build their Next Generation Multipurpose Microprocessor (NGMP) architecture, which is prototyped on the Xilinx XC5VFX130T FPGA board. However, the LEON architecture is fairly complex, and it is difficult to instantiate more than two or three on a medium-sized FPGA. Clack *et al* [4] investigate the use of FPGAs as a prototyping platform for developing multicore system applications. They use the Xilinx MicroBlaze processor for the core, and a bus protocol for the inter-core communication. Some designs focus primarily on the Network-on-chip (NoC). Lusala *et al* [13], for example, propose a scalable implementation of NoC on FPGA using a torus topology. Genko *et al* [8] also present an FPGA-based flexible emulation environment for exploring different NoC features. A VHDL-based cycle-accurate RTL model for evaluating power and performance of NoC architectures is presented in Banerjee *et al* [2]. Other designs make use of multiple FPGAs. H-Scale [19], by Saint-Jean *et al*, is a multi-FPGA based homogeneous SoC, with RISC processors and an asynchronous NoC. The S-Scale version supports a multi-threaded sequential programming model with dedicated communication primitives handled at runtime by a simple operating system.

3. HERACLES HARDWARE SYSTEM

Heracles presents designers with a global and complete view of the inner workings of the multi/many-core system at cycle-level granularity from instruction fetches at the processing core in each node to the flit arbitration at the routers. It enables designers to explore different implementation parameters: core micro-architecture, levels of caches, cache sizes, routing algorithm, router micro-architecture, distributed or shared memory, or network interface, and to quickly evaluate their impact on the overall system performance. It is implemented with user-enabled performance counters and probes.

3.1 System overview

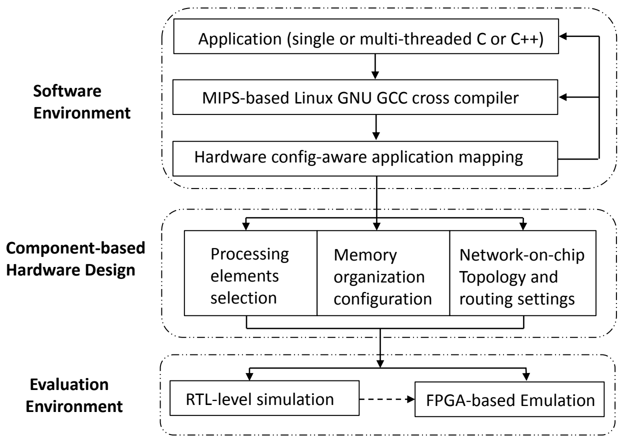
Software Environment
Application (single or multi-threaded C or C++)
MIPS-based Linux GNU GCC cross compiler
Hardware config-aware application mapping
Component-based Hardware Design
Processing elements selection
Memory organization configuration
Network-on-chip Topology and routing settings
Evaluation Environment
RTL-level simulation
FPGA-based Emulation

Figure 1: *Heracles*-based design flow.

Figure 1 illustrates the general *Heracles*-based design flow. Full applications–written in single or multithreaded C or C++–can be directly compiled onto a given system instance using the *Heracles* MIPS-based GCC cross compiler. The detailed compilation process and application examples are presented in Section 5. For a multi/many-core system, we take a component-based approach by providing clear interfaces to all modules for easy composition and substitutions. The system has multiple default settings to allow users to quickly get a system running and only focus on their area of interest. System and application binary can be executed in an RTL simulated environment and/or on an FPGA. Figure 2 shows two different views of a typical network node structure in *Heracles.*

3.2 Processing Units

In the current version of the *Heracles* design framework, users can instantiate four different types of processor cores, or any combination thereof, depending on the programming model adopted and architectural evaluation goals.

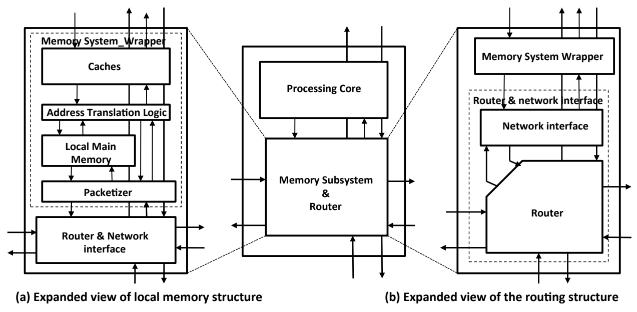
Memory System_Wrapper
Caches
Address Translation Logic
Local Main Memory
Packetizer
Router & Network interface
Processing Core
Memory Subsystem & Router
Memory System Wrapper
Router & network interface
Network interface
Router
(a) Expanded view of local memory structure
(b) Expanded view of the routing structure

Figure 2: Network node structure.

3.2.1 Injector Core

The injector core (iCore) is the simplest processing unit. It emits and/or collects from the network user-defined data streams and traffic patterns. Although it does not do any useful computation, this type of core is useful when the user is only focusing on the network on-chip behavior. It is useful in generating network traffic and allowing the evaluation of network congestion. Often, applications running on real cores fail to produce enough data traffic to saturate the network.

3.2.2 Single Hardware-Threaded MIPS Core

This is an integer 7-stage 32-bit MIPS–Microprocessor without Interlocked Pipeline Stages–Core (sCore). This RISC architecture is widely used in commercial products and for teaching purposes [17]. Most users are very familiar with this architecture and its operation, and will be able to easily modify it when necessary. Our implementation is generally standard with some modifications for FPGAs. For example, the adoption of a 7-stage pipeline, due to block RAM access time on the FPGA. The architecture is fully bypassed, with no branch prediction table or branch delay slot, running MIPS-III instruction set architecture (ISA) without floating point. Instruction and data caches are implemented using block RAMs, and instruction fetch and data memory access take two cycles. Stall and bypass signals are modified to support the extended pipeline. Instructions are issued and executed in-order, and the data memory accesses are also in-order.

3.2.3 Two-way Hardware-Threaded MIPS Core

A fully functional fine-grain hardware multithreaded MIPS core (dCore). There are two hardware threads in the core. The execution datapath for each thread is similar to the single-threaded core above. Each of the two threads has its own context which includes a program counter (PC), a set of 32 data registers, and one 32-bit state register. The core can dispatch instructions from any one of hardware contexts and supports precise interrupts (doorbell type) with limited state saving. A single hardware thread is active on any given cycle, and pipeline stages must be drained between context switches to avoid state corruption. The user has the ability to control the context switching conditions, e.g., minimum number of cycles to allocate to each hardware thread at a time, instruction or data cache misses.

3.2.4 Two-way Hardware-Threaded MIPS Core with Migration

The fourth type of core is also a two-way hardware-threaded processor but enhanced to support hardware-level thread migration and evictions (mCore). It is the user's responsibility to guarantee deadlock-freedom under this core configuration. One approach is to allocate local memory to contexts so on migration they are removed from the network. Another approach which requires no additional hardware modification to the core, is using Cho et al [3] deadlock-free thread migration scheme.

3.2.5 FPGA Synthesis Data

All the cores have the same interface, they are self-contained and oblivious to the rest of the system, and therefore easily interchangeable. The cores are synthesized using Xilinx ISE Design Suite 11.5, with Virtex-6 LX550T package ff1760 speed -2, as the targeted FPGA board. The number of slice registers and slice lookup tables (LUTs) on the board are 687360 and 343680 respectively. Table 1 shows the register and LUT utilization of the different cores. The two-way hardware-threaded core with migration consumes the most resources and is less than 0.5%. Table 1 also shows the clocking speed of the cores. The injector core, which does no useful computation, runs the fastest at $500.92MHz$ whereas the two-way hardware-threaded core runs the slowest at $118.66MHz$.

Table 1: FPGA resource utilization per core type.

Core type	iCore	sCore	dCore	mCore
Registers	227	1660	2875	3484
LUTs	243	3661	5481	6293
Speed (MHz)	500.92	172.02	118.66	127.4

3.3 Memory System Organization

The memory system in *Heracles* is parameterized, and can be set up in various ways, independent of the rest of the system. The key components are main memory, caching system, and network interface.

3.3.1 Main Memory Configuration

The main memory is constructed to allow different memory space configurations. For Centralized Shared Memory (CSM) implementation, all processors share a single large main memory block; the local memory size (shown in Figure 2) is simply set to zero at all nodes except one. In Distributed Shared Memory (DSM), where each processing element has a local memory, the local memory is parameterized and has two very important attributes: the size can be changed on a per core-basis, providing support for both uniform and non-uniform distributed memory, and it can service a variable number of caches in a round-robin fashion. Figure 3 illustrates these physical memory partitions. The fact that the local memory is parameterized to handle requests from a variable number of caches allows the traffic coming into a node from other cores through the network to be presented to local memory as just another cache communication. This illusion is created through the network packetizer. Local memory can also be viewed as a memory controller. Figure 4 illustrates the local structure of the memory sub-system. The *LOCAL_ADDR_BITS* parameter is used to set the size of the local memory. The *Address Translation Logic* performs the virtual-to-physical address lookup using the high-order bits, and directs cache traffic to local memory or network.

For cache coherence, a directory is attached to each local memory and the MESI protocol is implemented as the

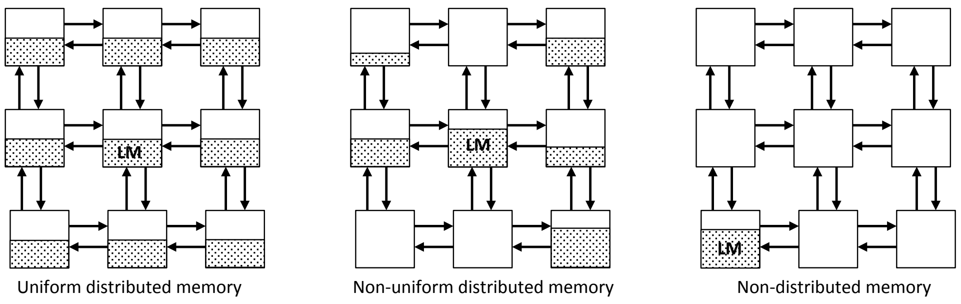

Figure 3: Possible physical memory configurations.

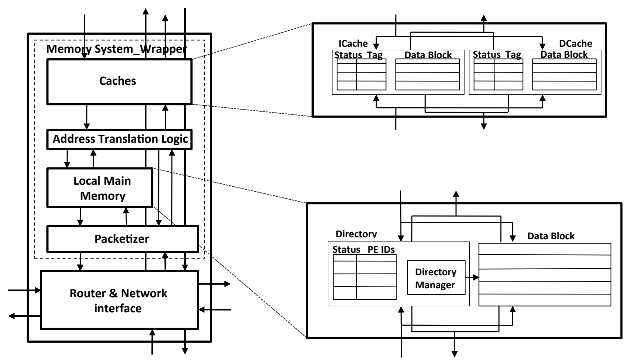

Figure 4: Local memory sub-system structure.

default coherence mechanism. Remote access (RA) is also supported. In RA mode, the network packetizer directly sends network traffic to the caches. Memory structures are implemented in FPGA using block RAMs. There are 632 block RAMs on the Virtex-6 LX550T. A local memory of $0.26MB$ uses 64 block RAMs or 10%. Table 2 shows the FPGA resource used to provide the two cache coherence mechanisms. The RA scheme uses less hardware resources than the cache-coherence-free structure, since no cache-line buffering is needed. The directory-based coherence is far more complex resulting in more resource utilization. The *SHARERS* parameter is used to set the number of sharers per data block. It also dictates the overall size of the local memory directory size. When a directory entry cannot handle all sharers, other sharers are evicted.

Table 2: FPGA resource utilization per coherence mechanism.

Coherence	None	RA	Directory
Registers	2917	2424	11482
LUTs	5285	4826	17460
Speed (MHz)	238.04	217.34	171.75

3.3.2 Caching System

The user can instantiate direct-mapped Level 1 or Levels 1 and 2 caches with the option of making Level 2 an inclusive cache. The *INDEX_BITS* parameter defines the number of blocks or cache-lines in the cache where the *OFFSET_BITS* parameter defines block size. By default, cache and memory structures are implemented in FPGA using block RAMs, but user can instruct *Heracles* to use LUTs for caches or some combination of LUTs and block RAMs. A single $2KB$ cache uses 4 FPGA block RAMs, 462 slice registers, 1106 slice LUTs, and runs at $228.8MHz$. If cache size is increased to $8KB$ by changing the *INDEX_BITS* parameter from 6 to 8, resource utilization and speed remain identical. Meanwhile if cache size is increased to $8KB$ by changing the *OFFSET_BITS* parameter from 3 to 5, resource utilization increases dramatically: 15 FPGA block RAMs, 1232 slice registers, 3397 slice LUTs, and speed is $226.8MHz$. FPGA-based cache design favors large number of blocks of small size versus small number of blocks of large size [1].

3.3.3 Network Interface

The *Address Resolution Logic* works with the *Packetizer* module, shown in Figure 2, to get the caches and the local memory to interact with the rest of the system. All cache traffic goes through the *Address Resolution Logic*, which determines if a request can be served at the local memory, or if the request needs to be sent over the network. The *Packetizer* is responsible for converting data traffic, such as a load, coming from the local memory and the cache system into packets or flits that can be routed inside the Network-on-chip (NoC), and for reconstructing packets or flits into data traffic at the opposite side when exiting the NoC.

3.3.4 Hardware multithreading and caching

In this section, we examine the effect of hardware multithreading (HMT) on system performance. We run the 197.parser application from the SPEC CINT2000 benchmarks on a single node with the dCore as the processing unit using two different inputs–one per thread–with five different execution interleaving policies:

- setup 1: threads take turns to execute every 32 cycles; on a context switch, the pipeline is drained before the execution of another thread begins.
- setup 2: thread switching happens every 1024 cycles.
- setup 3: thread context swapping is initiated on an instruction or a data miss at the Level 1 cache.
- setup 4: thread interleaving occurs only when there is a data miss at the Level 1 cache.
- setup 5: thread switching happens when there is a data miss at the Level 2 cache.

Figure 5 shows the total completion time of the two threads (in terms of number of cycles). It is worth noting that even with fast fine-grain hardware context switching, multithreading is most beneficial for large miss penalty events like Level 2 cache misses or remote data accesses.

3.4 Network-on-Chip (NoC)

To provide scalability, *Heracles* uses a network-on-chip (NoC) architecture for its data communication infrastructure. A NoC architecture is defined by its topology (the

[1]Cache-line size also has traffic implications at the network level

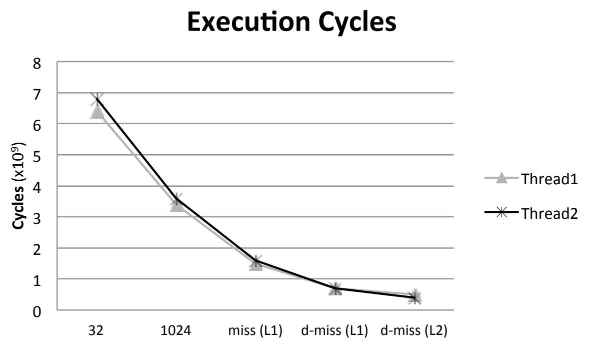

Figure 5: Effects of hardware multithreading and caching.

physical organization of nodes in the network), its flow control mechanism (which establishes the data formatting, the switching protocol and the buffer allocation), and its routing algorithm (which determines the path selected by a packet to reach its destination under a given application).

3.4.1 Flow control

Routing in *Heracles* can be done using either bufferless or buffered routers. Bufferless routing is generally used to reduce area and power overhead associated with buffered routing. Contention for physical link access is resolved by either dropping and retransmitting or temporarily misrouting or *deflecting* of flits. With flit dropping an acknowledgment mechanism is needed to enable retransmission of lost flits. With flit deflection, a priority-based arbitration, e.g., *age-based*, is needed to avoid livelock. In *Heracles*, to mitigate some of the problems associated with the lossy bufferless routing, namely retransmission and slow arbitration logic, we supplement the arbiter with a routing table that can be statically and off-line configured on a per-application basis.

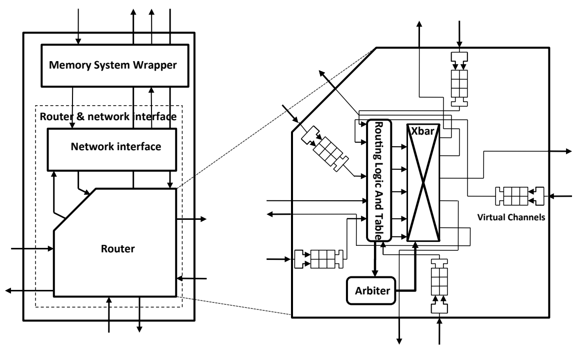

Figure 6: Virtual channel based router architecture.

The system default virtual-channel router conforms in its architecture and operation to conventional virtual-channel routers [5]. It has some input buffers to store flits while they are waiting to be routed to the next hop in the network. The router is modular enough to allow user to substitute different arbitration schemes. The routing operation takes four steps or phases, namely routing (RC), virtual-channel allocation (VA), switch allocation (SA), and switch traversal (ST), where each phase corresponds to a pipeline stage in our router. Figure 6 depicts the general structure of the buffered router. In this router the number of virtual channels per port and their sizes are controlled through *VC_PER_PORT* and *VC_DEPTH* parameters. Table 3 shows the register and LUT utilization of the bufferless router and different buffer configurations of the buffered router. It also shows the effect of virtual channels on router clocking speed. The key take-away is that a larger number of VCs at the router increases both the router resource utilization and the critical path.

Table 3: FPGA resource utilization per router configuration.

Number of VCs	Bufferless	2 VCs	4 VCs	8 VCs
Registers	175	4081	7260	13374
LUTs	328	7251	12733	23585
Speed (MHz)	817.18	111.83	94.8	80.02

3.4.2 Routing algorithm

Algorithms used to compute routes in network-on-chip (NoC) architectures, generally fall under two categories: *oblivious* and *dynamic* [16]. The default routers in *Heracles* primarily support *oblivious* routing algorithms using either fixed logic or routing tables. Fixed logic is provided for dimension-order routing (DOR) algorithms, which are widely used and have many desirable properties. On the other hand, table-based routing provides greater programmability and flexibility, since routes can be pre-computed and stored in the routing tables before execution. Both buffered and bufferless routers can make usage of the routing tables. *Heracles* provides support for both static and dynamic virtual channel allocation.

3.4.3 Network Topology Configuration

The parameterization of the number of input ports and output ports on the router and the table-based routing capability give *Heracles* a great amount of flexibility and the ability to metamorphose into different network topologies; for example, *k*-ary *n*-cube, 2D-mesh, 3D-mesh, hypercube, ring, or tree. A new topology is constructed by changing the *IN_PORTS*, *OUT_PORTS*, and *SWITCH_TO_SWITCH* parameters and reconnecting the routers. Table 4 shows the clocking speed of a bufferless router, a buffered router with strict round-robin arbitration (Arbiter1), a buffered router with weak round-robin arbitration (Arbiter2), and a buffered router with 7 ports for a 3D-mesh network. The bufferless router runs the fastest at $817.2MHz$, Arbiter1 and Arbiter2 run at the same speed (~ 112), although the arbitration scheme in Arbiter2 is more complex. The 7-port router runs the slowest due to more complex arbitration logic.

Table 4: Clocking speed of different router types.

Router type	Bufferless	Arbiter1	Arbiter2	7-Port
Speed (MHz)	817.18	111.83	112.15	101.5

Figure 7 shows a 3×3 2D-mesh with all identical routers. Figure 8 depicts an unbalanced *fat-tree* topology. For a *fat-tree* [11] topology, routers at different levels of the tree have different sizes, in terms of crossbar and arbitration logic. The root node contains the largest router, and controls the clock frequency of the system.

4. HERACLES PROGRAMMING MODELS

A programming model is inherently tied to the underlying hardware architecture. The *Heracles* design tool has no directly deployable operating system, but it supports both sequential and parallel programming models through various

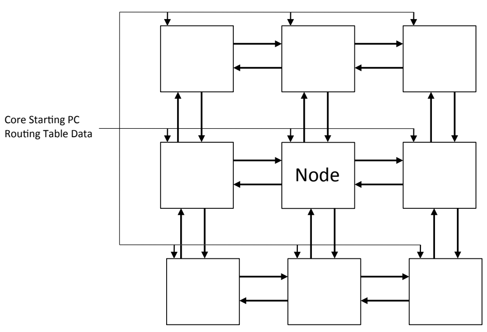

Figure 7: 2D mesh topology.

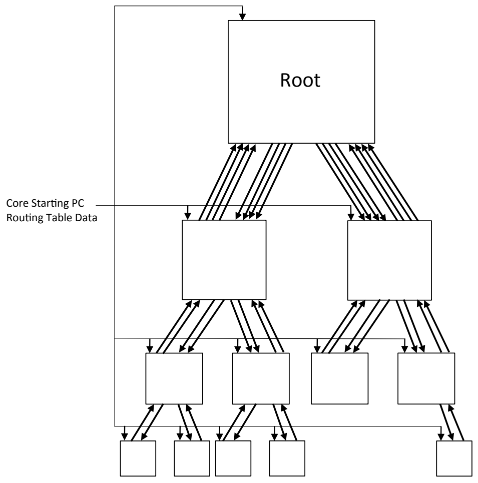

Figure 8: Unbalanced fat-tree topology.

application programming interface (API) protocols. There are three memory spaces associated with each program: instruction memory, data memory, and stack memory. Dynamic memory allocation is not supported in the current version of the tool.

4.1 Sequential programming model

In the sequential programming model, a program has a single entry point (starting program counter–PC) and single execution thread. Under this model, a program may exhibit any of the follow behavior or a combination thereof:

- the local memory of executing core has instruction binary;
- PC pointer to another core's local memory where the instruction binary resides;
- the stack frame pointer– SP points to the local memory of executing core;
- SP points to another core's local memory for the storage of the stack frame;
- program data is stored at the local memory;
- program data is mapped to another core local memory.

Figure 9 gives illustrating examples of the memory space management when dealing with sequential programs. These techniques provide the programming flexibility needed to support the different physical memory configurations. They also allow users:

- to run the same program on multiple cores (program inputs can be different); in this setup the program binary is loaded to one core and the program counter at other cores points to the core with the binary;

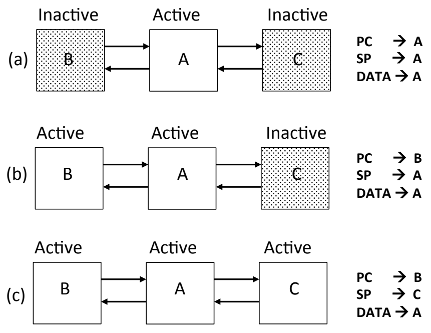

Figure 9: Examples of memory space management for sequential programs.

- to execute one program across multiple cores by migrating the program from one core to another.

4.2 Parallel programming model

The *Heracles* design platform supports both hardware multi-threading and software multi-threading. The keywords *HHThread1* and *HHThread2* are provided to specify the part of the program to execute on each hardware thread. Multiple programs can also be executed on the same core using the same approach. An example is shown below:

```
...
int HHThread1 (int *array, int item, int size) {
    sort(array, size);
    int output = search(array, item, size);
    return output;
}

int HHThread2 (int input,int src,int aux,int dest){
    int output = Hanoi(input, src, aux, dest);
    return output;
}
...
```

Below is the associated binary outline:

```
@e2       // <HHThread1>
27bdffd8  // 00000388 addiu sp,sp,-40
...
0c00004c  // 000003c0 jal 130 <search>
...

@fa       // <HHThread2>
27bdffd8  // 000003e8 addiu sp,sp,-40
...
0c0000ab  // 00000418 jal 2ac <Hanoi>
...

@110      // <HHThread1_Dispatcher>
27bdffa8  // 00000440 addiu sp,sp,-88
...
0c0000e2  // 000004bc jal 388 <hThread1>
...

@15e      // <HHThread2_Dispatcher>
27bdffa8  // 00000440 addiu sp,sp,-88
...
0c0000fa  // 000004d8 jal 3e8 <HHThread2>
...
```

Heracles uses OpenMP style pragmas to allow users to directly compile multi-threaded programs onto cores. Users specify programs or parts of a program that can be executed in parallel and on which cores. Keywords *HLock* and *HBarrier* are provided for synchronization and shared variables are encoded with the keyword *HGlobal*. An example is shown below:

```
...
#pragma Heracles core 0 {
   // Synchronizers
   HLock lock1,  lock2;
   HBarrier bar1, bar2;

   // Variables
   HGlobal int arg1, arg2, arg3;
   HGlobal int Arr[16][16];
   HGlobal int Arr0[16][16] = { { 1, 12, 7, 0,...
   HGlobal int Arr1[16[16] = { { 2, 45, 63, 89,...

   // Workers
   #pragma Heracles core 1 {
      start_check(50);
      check_lock(&lock1, 1000);
      matrix_add(Arr, Arr0, Arr1, arg1);
      clear_barrier(&bar1);
   }

   #pragma Heracles core 2 {
      start_check(50);
      check_lock(&lock1, 1000);
      matrix_add(Arr, Arr0, Arr1, arg2);
      clear_barrier(&bar2);
   }
}
 ...
```

Below is the intermediate C representation:

```
 ...
int core_0_lock1, core_0_lock2;
int core_0_bar1, core_0_bar2;
int core_0_arg1, core_0_arg2, core_0_arg3;
int core_0_Arr[16][16];
int core_0_Arr0[16][16] = { { 1, 12, 7, 0,...
int core_0_Arr1[16][16] = { { 2, 45, 63, 89,...

void  core_0_work (void)
{
   // Synchronizers
   // Variables
   // Workers
   // Main function
   main();
}
void  core_1_work (void)
{
   start_check(50);
   check_lock(&core_0_lock1, 1000);
   matrix_add(core_0_Arr, core_0_Arr0, core_0_Arr1,
   core_0_arg1);
   clear_barrier(&core_0_bar1);
}
...
```

Below is the associated binary outline:

```
@12b      // <Dispatcher>
27bdffe8  // 000004ac addiu sp,sp,-24
...
0c000113  // 000004bc jal 44c <core_0_work>
...
```

5. PROGRAMMING TOOLCHAIN

5.1 Program compilation flow

The *Heracles* environment has an open-source compiler toolchain to assist in developing software for different system configurations. The toolchain is built around the GCC MIPS cross-compiler using GNU C version 3.2.1. Figure 10 depicts the software flow for compiling a C program into the compatible MIPS instruction code that can be executed on the system. The compilation process consists of a series of six steps.

- First, the user invokes *mips-gcc* to translate the C code into assembly language (e.g., *./mips-gcc -S fibonacci.c*).
- In step 2, the assembly code is then run through the isa-checker (e.g., *./checker fibonacci.s*). The checker's role is to: (1) remove all memory space primitives, (2) replace all pseudo-instructions, and (3) check for floating point instructions. Its output is a *.asm* file.
- For this release, there is no direct high-level operating system support. Therefore, in the third compilation stage, a small kernel-like assembly code is added to the application assembly code for memory space management and workload distribution (e.g., *./linker fibonacci.asm*). Users can modify the *linker.cpp* file provided in the toolchain to reconfigure the memory space and workload.
- In step 4, the user compiles the assembly file into an object file using the cross-compiler. This is accomplished by executing *mips-as* on the *.asm* file (e.g., *./mips-as fibonacci.asm*).
- In step 5, the object file is disassembled using the *mips-objdump* command (e.g., *./mips-objdump fibonacci.o*). Its output is a *.dump* file.
- Finally, the constructor script is called to transform the dump file into a Verilog memory, *.vmh*, file format (e.g., *./dump2vmh fibonacci.dump*).

If the program is specified using *Heracles* multi-threading format (*.hc* or *.hcc*), c-cplus-generator (e.g., *./c-cplus-generator fibonacci.hc*) is called to first get the C or C++ program file before executing the steps listed above. The software toolchain is still evolving. All these steps are also automated through the GUI.

5.2 Heracles graphical user interface

The graphical user interface (GUI) is called *Heracles Designer*. It helps to quickly configure and launch system configurations. Figure 11 shows a screen shot of the GUI. On the core tab, the user can select: (1) the type of core to generate, (2) the network topology of the system instance to generate, (3) the number of cores to generate, (4) traffic type, injection rate, and simulation cycles in the case of an injector core, or (5) different pre-configured settings.

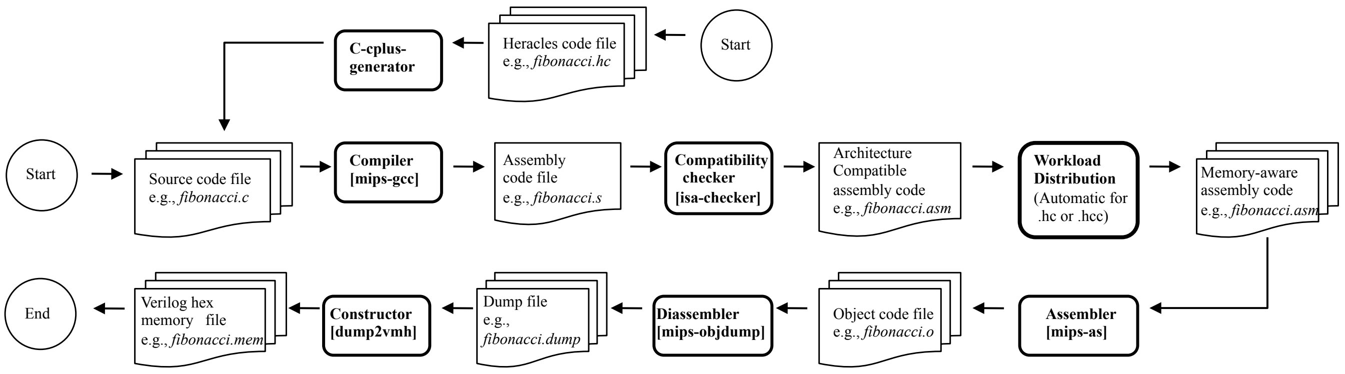

Figure 10: Software toolchain flow.

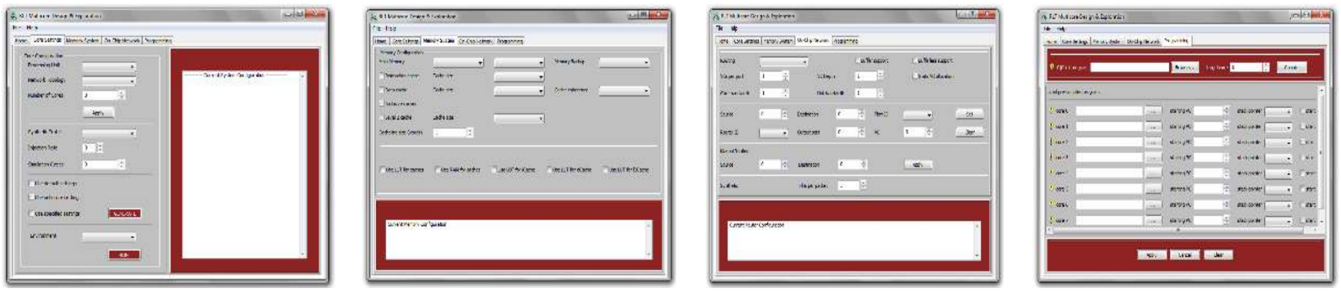

Figure 11: *Heracles* designer graphical user interface.

Generate and *Run* buttons on this tab are used to automatically generate the Verilog files and to launch the synthesis process or specified simulation environment. The second tab–memory system tab–allows the user to set: (1) maim memory configuration (e.g., Uniformed Distributed), (2) total main memory size, (3) instruction and data cache sizes, (4) Level 2 cache, and (5) FPGA favored resource (LUT or block RAM) for cache structures. The on-chip network tab covers all the major aspects of the system interconnect: (1) routing algorithm, (2) number of virtual channels (VCs) per port, (3) VC depth, (4) core and switch bandwidths, (5) routing tables programming, by selecting source/destination pair or flow ID, router ID, output port, and VC (allowing user-defined routing paths), and (6) number of flits per packet for injector-based traffic. The programming tab is updated when the user changes the number of cores in the system; the user can: (1) load a binary file onto a core, (2) load a binary onto a core and set the starting address for another core to point to that binary, (3) select where to place the data section or stack pointer of a core (it can be local, on the same core as the binary or on another core), and (4) select which cores to start.

6. EXPERIMENTAL RESULTS

6.1 Full 2D-mesh systems

The synthesis results of five multicore systems of size: 2×2, 3×3, 4×4, 5×5, and 6×6 arranged in 2D-mesh topology are summarized below. Table 5 gives the key architectural characteristics of the multicore system. All five systems run at 105.5MHz, which is the clock frequency of the router, regardless of the size of the mesh.

Figure 12 summarizes the FPGA resource utilization by the different systems in terms of registers, lookup tables, and block RAMs. In the 2×2 and 3×3 configurations, the local memory is set to 260KB per core. The 3×3 configuration uses 99% of block RAM resources at 260KB of local memory per core. For the 4×4 configuration the local memory is reduced to 64KB per core, and the local memory in the 5×5 configuration is set to 32KB. The 6×6 configuration, with 16KB of local memory per core, fails during the mapping and routing synthesis steps, due to the lack of LUTs.

6.2 Evaluation results

We examine the performance of two SPEC CINT2000 benchmarks, namely, 197.parser and 256.bzip2 on *Heracles*. We modify and parallelize these benchmarks to fit into our evaluation framework. For the 197.parser benchmark, we identify three functional units: file reading and parameters setting as one unit, actual parsing as a second unit, and er-

Table 5: 2D-mesh system architecture details.

Core	
ISA	32-Bit MIPS
Hardware threads	1
Pipeline Stages	7
Bypassing	Full
Branch policy	Always non-Taken
Outstanding memory requests	1
Level 1 Instruction/Data Caches	
Associativity	Direct
Size	variable
Outstanding Misses	1
On-Chip Network	
Topology	2D-Mesh
Routing Policy	DOR and Table-based
Virtual Channels	2
Buffers per channel	8

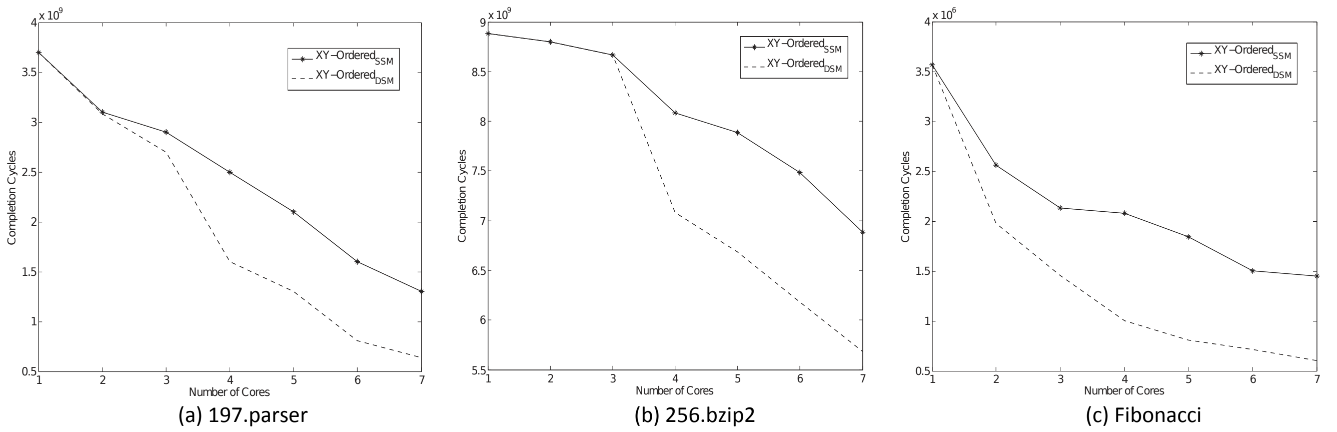

Figure 13: Effect of memory organization on performance for the different applications.

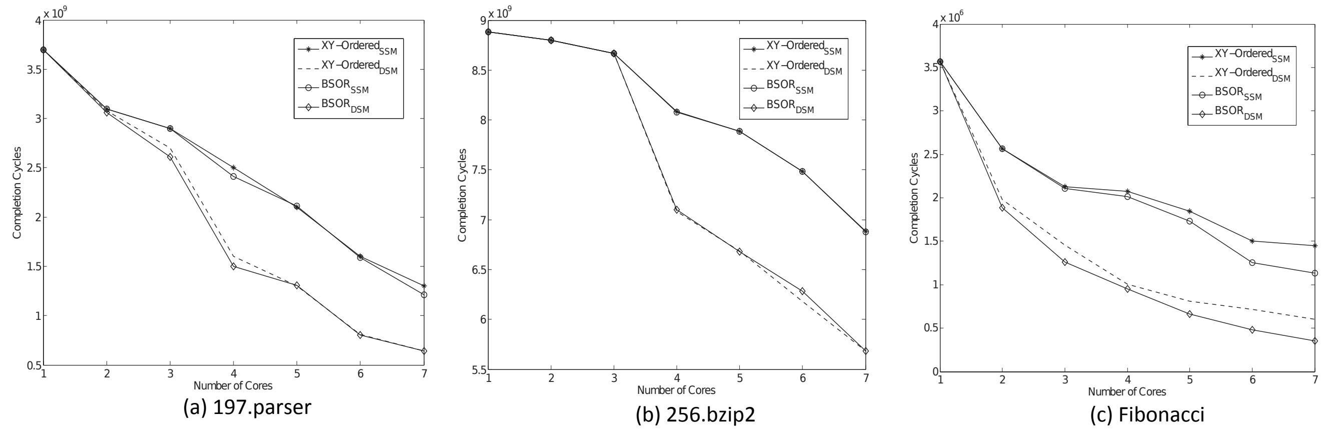

Figure 14: Effect of routing algorithm on performance in 2D-mesh for the different applications.

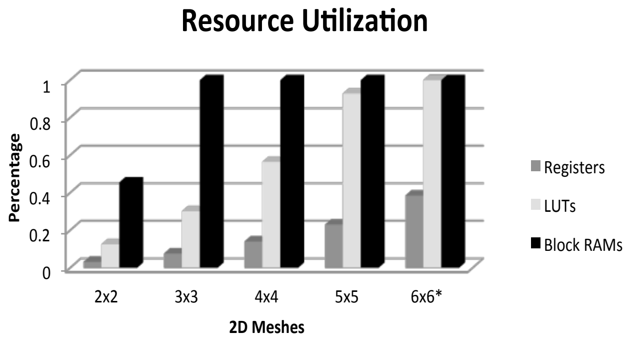

Figure 12: Percentage of FPGA resource utilization per mesh size.

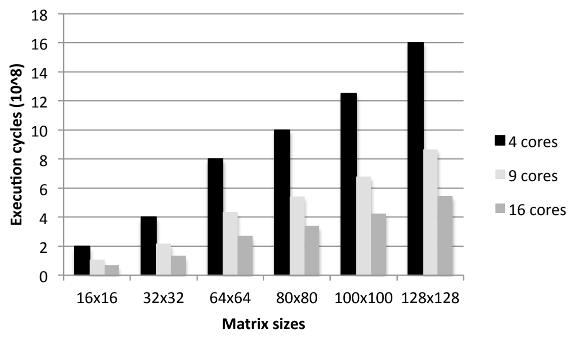

Figure 15: Matrix multiplication acceleration.

ror reporting as the third unit. When there are more than three cores, all additional cores are used in the parsing unit. Similarly, 256.bzip2 is divided into three functional units: file reading and cyclic redundancy check, compression, and output file writing. The compression unit exhibits a high degree of data-parallelism, therefore we apply all additional cores to this unit for core count greater than three. We also present a brief analysis of a simple Fibonacci number calculation program. Figures 13 (a), (b), and (c) show 197.parser, 256.bzip2, and Fibonacci benchmarks under single shared-memory (SSM) and distributed shared-memory (DSM), using *XY-Ordered* routing. Increasing the number of cores improves performance for both benchmarks; it also exposes the memory bottleneck encountered in the single shared-memory scheme. Figures 14 (a), (b), and (c) highlight the impact of the routing algorithm on the overall system performance, by comparing completion cycles of *XY-Ordered* routing and BSOR [9]. BSOR, which stands for Bandwidth-Sensitive Oblivious Routing, is a table-based routing algorithm that minimizes the maximum channel load (MCL) (or maximum traffic) across all network links in an effort to maximize application throughput. The routing algorithm has little or no effect on the performance of 197.parser and 256.bzip2 benchmarks, because of the traffic patterns in these applications. For the Fibonacci application, Figure 14 (c), BSOR routing does improve performance, particularly with 5 or more cores. To show the multithreading and scalability properties of the system, Figure 15 presents the execution times for matrix multiplication given matrices of different size. The *Heracles* multicore programming format is used to automate of the workload distribution onto cores.

7. CONCLUSION

In this work, we present the new *Heracles* design toolkit which is comprised of soft hardware (HDL) modules, an application compiler toolchain, and a graphical user interface. It is a component-based framework that gives researchers the ability to create complete, realistic, synthesizable, multi/many-core architectures for fast, high-accuracy design space exploration. In this environment, user can explore design tradeoffs at the processing unit level, the memory organization and access level, and the network on-chip level. The *Heracles* tool is open-source and can be downloaded at http://projects.csail.mit.edu/heracles. In the current release, RTL hardware modules can be simulated on all operating systems, the MIPS GCC cross-compiler runs in a Linux environment, and the graphical user interface has a Windows installer.

8. ACKNOWLEDGMENTS

We thank the students from the MIT 6.S918 IAP 2012 class who worked with early versions of the tool and made many excellent suggestions for improvement. We thank Joel Emer, Li-Shiuan Peh, and Omer Khan for interesting discussions throughout the course of this work.

9. REFERENCES

[1] J. Andersson, J. Gaisler, and R. Weigand. Next generation multipurpose microprocessor. 2010.

[2] N. Banerjee, P. Vellanki, and K. Chatha. A power and performance model for network-on-chip architectures. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1250 – 1255 Vol.2, feb. 2004.

[3] M. H. Cho, K. S. Shim, M. Lis, O. Khan, and S. Devadas. Deadlock-free fine-grained thread migration. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pages 33 –40, may 2011.

[4] C. R. Clack, R. Nathuji, and H.-H. S. Lee. Using an fpga as a prototyping platform for multi-core processor applications. In *WARFP-2005: Workshop on Architecture Research using FPGA Platforms*, Cambridge, MA, USA, feb. 2005.

[5] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[6] P. Del valle, D. Atienza, I. Magan, J. Flores, E. Perez, J. Mendias, L. Benini, and G. Micheli. A complete multi-processor system-on-chip fpga-based emulation framework. In *Very Large Scale Integration, 2006 IFIP International Conference on*, pages 140–145, oct. 2006.

[7] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer. Leveraging latency-insensitivity to ease multiple fpga design. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 175–184, New York, NY, USA, 2012.

[8] N. Genko, D. Atienza, G. De Micheli, J. Mendias, R. Hermida, and F. Catthoor. A complete network-on-chip emulation framework. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 246–251 Vol. 1, march 2005.

[9] M. Kinsy, M. H. Cho, T. Wen, E. Suh, M. van Dijk, and S. Devadas. Application-Aware Deadlock-Free Oblivious Routing. In *Proceedings of the Int'l Symposium on Computer Architecture*, June 2009.

[10] M. Kinsy, M. Pellauer, and S. Devadas. Heracles: Fully synthesizable parameterized mips-based multicore system. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 356 –362, sept. 2011.

[11] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, 1985.

[12] M. Lis, P. Ren, M. H. Cho, K. S. Shim, C. Fletcher, O. Khan, and S. Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 175–185, april 2011.

[13] A. Lusala, P. Manet, B. Rousseau, and J.-D. Legat. Noc implementation in fpga using torus topology. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 778–781, aug. 2007.

[14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, feb 2002.

[15] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, jan. 2010.

[16] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[17] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[18] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 406–417, feb. 2011.

[19] N. Saint-Jean, G. Sassatelli, P. Benoit, L. Torres, and M. Robert. Hs-scale: a hardware-software scalable mp-soc architecture for embedded systems. In *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, pages 21–28, march 2007.

[20] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovict' and. Ramp gold: An fpga-based architecture simulator for multiprocessors. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 463–468, june 2010.

[21] W. Yu. Gems a high performance em simulation tool. In *Electrical Design of Advanced Packaging Systems Symposium, 2009. (EDAPS 2009). IEEE*, pages 1–4, dec. 2009.