DECADES Documentation and Specifications

A Introduction

Below we provide in-depth descriptions about the DECADES compiler and the DECADES simulator. We have additionally included links to video demos of our tools.

B Compiler

This section describes the DECADES compiler, DEC++.

At a high level, DEC++ is a convenient wrapper around a collection of Clang visitor and LLVM transformation passes. DEC++ can be invoked directly on a set of `.cpp` source files and mirrors the behavior of a traditional C++ compiler, e.g. g++. Using various compiler *modes*, the source code can be compiled and transformed to target various features of the DECADES architecture. The output of DEC++ can be one of: (1) an x86 native executable, (2) LLVM computations graphs for the Pythia simulator (see Section C), or (3) a RISC-V binary to execute on our target chip architecture composed of OpenPiton and Ariane.

B.1 Annotations and Limitations

We begin by discussion annotations and limitations of the compiler for C++ programs. While the list may seem daunting, we have found that for many applications of interest (including the SDH target applications) these restrictions and annotations have not been an issue as they are straightforward to incorporate.

- **Kernels:** DEC++ targets a single kernel function. This function must be named `_kernel_` and have a return type of `void`.
- **Tile ID and Tile Count:** Tiles are the base computation model for DECADES. Therefore, the last two arguments to the `_kernel_` function must be two integers that represent a tile id for the kernel and the number of tiles. When invoking `_kernel_` (e.g. from `main`), the values for these arguments should be `DECADES_TILE_ID` and `DECADES_NUM_TILES`, both defined in the `DECADES/DECADES.h` header file that should be included. DEC++ will automatically populate these argument values based on the tile argument (discussed below).
- **Single Function:** DEC++ does not handle inter-function analysis. Thus, all functions called by the kernel are inlined in the first pass. The behavior of DEC++ is undefined if functions cannot be inlined. This includes code in external libraries. The compiler will warn, but not error, upon finding a function that it cannot inline.
- **I/O and Memory Allocation:** Kernel functions should not allocate memory or perform I/O operations. These should be done in the wrapper functions (e.g. main).
- **Recursion:** The source code cannot contain recursive functions, as these are not supported by DEC++.

B.2 DEC++ Modes

DEC++ has four different modes that it can run. These modes are invoked when DEC++ is run at the command line using the `--comp_model` (or `-m`) flag.

B.2.1 Native ("n"): The native mode simply compiles the source files with no LLVM transformations or passes. This mode is useful for verification of correct program output and debugging, as well as application development outside of any DECADES features.

B.2.2 DECADES Base ("db"): This is the default DEC++ mode. It identifies the `_kernel_` function, performs function inlining and wraps the function invocation in the tile launcher. This mode supports all three backend targets: x86 emulation, Pythia Simulator, and RISC-V binary.

B.2.3 Decoupled Implicit ("di"): This compilation mode slices the `_kernel_` function into supply and compute programs completely *automatically*. The supply and compute kernels communicate through a DECADES API that is included and linked in the compilation process. The compute slice does not access memory *at all*, and instead queries the supply unit for values.

The supply slice additionally goes through a second pass where *terminal* loads are identified and annotated so that can be handled in a special way by the architecture.

These API calls are *not* inlined, as the simulator can use them to explore architectural innovations. We provide a shared memory implementation, which allows for x86 emulation even without special hardware features.

B.2.4 Biscuit ("b"): This compilation mode assumes that the programmer has used our intelligent storage tile API (see Section B.5) in their program. The compiler then links to the library implementation and launches an intelligent storage tile to service the requests made by the API.

B.3 Tile Model and Single Program Multiple Thread (SPMT) Parallelism

The DEC++ programming model is tile based. Recall that the kernel function must take a tile id and tile count argument.

At compile time, the user can provide the tile count flag ("-t") to specify how many tiles to compile for. We denote this value as *N*. The user can then assume that: (1) *N* kernel functions will be run in parallel; (2) each kernel will be given a contiguous unique tile id, which is passed in the `DECADES_TILE_ID` argument; and (3) each kernel will receive the value N to fulfill the `DECADES_NUM_TILES` argument.

This provides an SPMT (single program, multiple thread) programming model by default. For programs that are not parallel, a value of 1 may be used for "-t", and the tile id and tile count arguments can safely be ignored (although integers must still be provided). The default value for "-t" is 1, so that users begin by programming a single tile.

B.3.1 Interaction with Modes:
- **Native:** The tile arguments are ignored for native mode.
- **DECADES Base:** The DECADES Nase mode can be used with tiles to provide N compute tiles, similar to a classic SPMT programming model (e.g. OpenMP).
- **Decoupled Implicit:** The DECADES Decoupled Implicit mode can also be used with tiles, which creates *N* compute tiles and *N* supply tiles that communicate pairwise.
- **Biscuit**: The tile arguments are currently ignored for biscuit mode.

B.3.2 Synchronization API: Communication across tiles is undefined except when using our synchronisation API. We currently support one instruction:
- `DECADES_BARRIER()`: An execution barrier that requires all tiles to reach the barrier and flush memory to a consistent view before continuing.

B.3.3 Implementation: The DECADES tile model is implemented using OpenMP under the hood. The first pass is a Clang visitor pass that duplicates kernel files for later slicing. It then wraps the kernel launch in an OpenMP environment and populates the tile id and tile count arguments.

B.4 Backends

DEC++ currently targets three backends. These backends are specified when DEC++ is run at the command line using the `--target` (or `-r`) flag.

B.4.1 x86 Emulation ("x86"): This backend creates an executable that can run natively on an x86 machine. Decoupled interactions, biscuit, and tiles are all emulated using shared memory. For decouling and biscuit, runtime information is gathered and printed at the end of the execution.

For the DECADES Decoupled Implicit mode, the displayed information consists of the number of interactions between the compute and supply slices, the types of data being transferred, and how many terminal loads were issued. For biscuit, the printed output includes the size of the communication buffer transferred between tiles and statistics about the vector widths of the computations passed to the biscuit tile. For the DECADES Base mode, the output is simply that of a shared memory SPMT program that can be used for performance benchmarking (e.g. against the SDH baselines).

All DEC++ modes and multiple tiles are supported for this backend.

B.4.2 Simulator ("simulator"): To target Pythia, the DECADES simulator, the user must provide the "-spp" (simulator preprocessing) flag with the path to the Pythia simulator preprocessing script (located in the simulator repository). This produces (1) instrumented x86 executables and (2) LLVM computation graphs. To run a DEC++ compiled application on the simulator, the instrumented x86 executables must be run to generate memory traces.

Once the memory traces have been created, the simulator can be run on the DEC++ output. For more details on the simulator, see Section C.

All modes except biscuit and native are supported for this backend. Multiple tiles are supported for this backend.

B.4.3 RISC-V ("riscv64"): This backend produces binaries to run on the multi-core RISC-V architecture composed of OpenPiton and Ariane, the hardware architecture of DECADES.

Currently only the DECADES Base mode is supported for this backend. Multiple tiles can be specified though, providing a SPMT programming target.

B.5 Intelligent Storage Tile API

Below are the functions a programmer can call to utilize an intelligent storage tile for parallelism and efficient memory accesses. Each intelligent storage tile contains a bit-serial processor ("biscuit"), which can quickly compute on large batches of parallel computations with SIMD instructions.

- `void` **`decades_ist_init()`** - initialize all intelligent storage tile memories
- `void` **`decades_ist_flag_set_char`**`(int` *`bid`*`, IS_compute_context*` *`local_ist_context`*`, char*` *`base`*`, unsigned long` *`offset`*`)` - set the value stored in the local scratchpad at *base[offset]* to be 1
  - *`bid`* = biscuit ID
  - *`local_ist_context`* = the scratchpad information (scratchpad and scratchpad size) local to the computation kernel
  - *`base`* = the array to store graph information
  - *`offset`* = the location in the array whose value is to be set to 1
- `void` **`decades_ist_incr_uint`**`(int` *`bid`*`, IS_compute_context*` *`local_ist_context`*`, unsigned int*` *`base`*`, unsigned long` *`offset`*`)` - increment the value stored in the local scratchpad at *base[offset]*
  - *`bid`* = biscuit ID
  - *`local_ist_context`* = the scratchpad information (scratchpad and scratchpad size) local to the computation kernel
  - *`base`* = the array to store graph information
  - *`offset`* = the location in the array whose value is to be incremented
- `void` **`decades_ist_flush`**`(int` *`bid`*`, IS_compute_context*` *`local_ist_context`*`)` - send batch of kernel computations to a specific biscuit in asynchronous biscuit mode
  - *`bid`* = biscuit ID
  - *`local_ist_context`* = the scratchpad information (scratchpad and scratchpad size) local to the computation kernel
- `void` **`decades_ist_load_uint_values`**`(int` *`bid`*`, unsigned int*` *`values`*`)` - load unsigned integer *values* into the local memory of biscuit with biscuit ID *bid*
  - *`bid`* = biscuit ID
  - *`values`* = the unsigned integer values to be stored in local memory
- `void` **`decades_ist_store_uint_values`**`(int` *`bid`*`, unsigned int*` *`values`*`)` - store unsigned integer *values* at the scratchpad addresses of biscuit with biscuit ID *bid*
  - *`bid`* = biscuit ID

  - `values` = the unsigned integer values to be stored in the scratchpad addresses
- `void` **`decades_ist_load_char_values`**`(int bid, char* values)` - load character *values* into the local memory of biscuit with biscuit ID *bid*
  - `bid` = biscuit ID
  - `values` = the character values to be stored in local memory
- `void` **`decades_ist_store_char_values`**`(int bid, char* values)` - store character *values* at the scratchpad addresses of biscuit with biscuit ID *bid*
  - `bid` = biscuit ID
  - `values` = the character values to be stored in the scratchpad addresses
- `void` **`decades_ist_finalize`**`(int bid)` - notify biscuit with biscuit ID *bid* that all computation has been completed so biscuit can shut down
  - `bid` = biscuit ID
- `void` **`print_ist_info()`** - print statistics collected (batch information, time spent waiting on compute and memory copies)

C Simulator

This section describes the API for the DECADES Simulator. Specifically, it provides guidelines that allow one to efficiently design accelerators that can be integrated together into the pre-existing framework. For information on building and running the simulator, see: `https://github.com/PrincetonUniversity/pythia` for the README.

C.1 Simulator Overview

The DECADES simulator is designed to be a timing simulator that provides performance estimates for the heterogeneous systems that appear in the DECADES architecture. Specifically, we hope to simulate applications leveraging multiple tiles (cores, accelerators, intelligent storage) running simultaneously.

C.2 The Digestor

At the center of the simulator is the *digestor*, which is responsible for instantiating all tiles, orchestrating their forward progress, and coordinating communication among them. It can also be thought of as a Network on Chip that receives packets (Transactions in this case) and sends them to their destination tile at the right cycle time for that tile. Hence, the goal of this API is to describe how every tile model must be designed to interface properly with it. The digestor is defined as the class Simulator and is found in pythia/sim/sim.h and pythia/sim/sim.cc. We show the interface functions a tile designer should be aware of below, but we discuss them in more depth in later sections.

```
//code snippet from pythia/sim/sim.h
void Simulator::InsertTransaction(Transaction* t, uint64_t cycles); //called by a tile to send
    transaction t to another tile (source and destination are contained in t).
void registerTile(Tile* tile, int tid); //this adds a tile to a simulator, explicitly assigning
    tile id tid to it
void registerTile(Tile* tile); //this adds a tile to the digestor, which auto assigns the tile an
    id
```

C.3 Tiles

Tiles run alongside each other, each being called upon by the digestor to take a one cycle step and increment its cycle count. Note that the tiles may run at different clock speeds; the digestor is aware of this and calls upon the tiles to step based on their provided clockspeeds. Tiles communicate by creating transactions and enqueuing them within the digestor, via a function call. The digestor is responsible for sending transactions to the destination tile at the right clock cycle for the destination tile, which it does by explicitly calling upon Tiles to receive messages.

C.4 Tile Abstract Class

To enforce the above behaviors, each tile must inherit the Tile abstract class in pythia/sim/tile/Tile.h, with the code shown below. It includes some virtual functions that must be implemented in any Tile subclass. See an example derived tile in pythia/sim/tile/tile_example/ExampleTile.cc. To run it and see the key highlights on how it was designed and the shell script to correctly set up the example files, follow the README at pythia/sim/tile/tile_example/README.

```
//code snippet from pythia/sim/tile/Tile.h
class Tile {
 public:
  int id; //tile id
  Simulator* sim; //the digestor
  uint64_t clockspeed; //clockspeed in MHertz
  uint64_t cycles; //cycle count

Tile(Simulator* sim, int clockspeed): sim(sim),clockspeed(clockspeed) {
    cycles=0;
  }
  virtual bool process()=0; //function to take a one-cycle simulation step, called by digestor
  virtual bool ReceiveTransaction(Transaction* t)=0; //function to receive a transaction, called
      by digestor
};
```

C.4.1 bool Tile::process(): This function is called by the digestor to force a tile to take a one-cycle step. Any implementation of it is required to do a cycle's worth of work, increase its cycle count, and return a boolean indicating if the tile could have remaining work (e.g., if it still could be invoked by other tiles or still has queued transactions it's processing). Note that more than a cycle's worth of work could be be done as long as the tile tracks when the results should be ready and acts on it when its cycle count has reached that future time. This is shown in pythia/sim/tile/tile_example/ExampleTile.cc.

C.4.2 bool Tile::ReceiveTransaction(Transaction* t): This function is called by the digestor to force a tile to receive a message. A boolean is returned to indicate if the message was successfully received or couldn't be received, perhaps due to resource constraints such as full message queues. The tile may chose to act immediately on a message pushed to it or enqueue it and process all its enqueued messages when the digestor calls process(). We will discuss the contents of *Transaction*, which represent message packets in Section C.5.1 below.

C.5 Inter-tile Communication

As discussed above, tiles receive messages through ReceiveTransaction() calls. However, if a tile wants to send a message to another tile, it simply calls sim->InsertTransaction(t, cycle), shown below. *t* is the transaction to be sent, and cycles is the clock cycle the tile intends the digestor to assume it was sent (cycle is typically the cycle count of the tile *cycles*).

Figure 1 illustrates inter-tile communication in which Tile0 wishes to communicate to Tile2, using Tile1 as an intermediary buffer. Tile0 sends the transaction t1 to the digestor at Tile0's cycle C1. The digestor waits until Tile1 reaches cycle C2 (the real-time equivalent of C1+some queue delay) and sends the transaction to Tile1. Tile 1 enqueues the transaction, modifies it to t2 and, after an appropriate number of cycles representing some queuing delay model internal to Tile1, sends the transaction to the digestor. Finally, the digestor relays the transaction to Tile2.

```
//code snippet from pythia/sim/tile/Tile.h
class Transaction {
public:
  Transaction(int id, int src_id, int dst_id) : id(id), src_id(src_id), dst_id(dst_id) {}; //
      constructor for inter-tile communication
  int id;
  int src_id;
  int dst_id;
  bool complete=false;
  TransType type=DEFAULT;
```

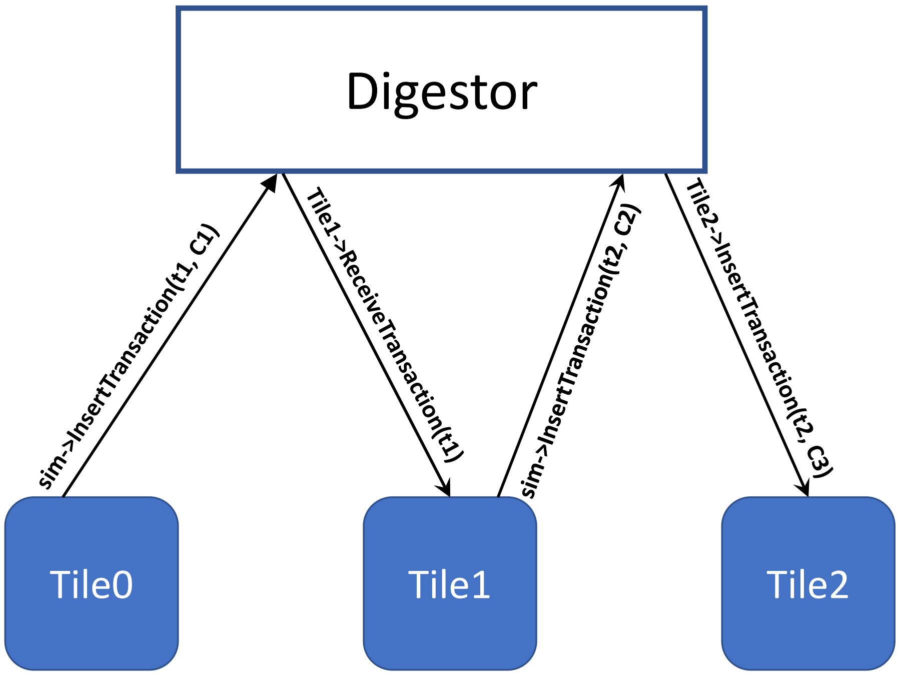
Digestor
sim->InsertTransaction(t1, C1)
Tile1->ReceiveTransaction(t1)
sim->InsertTransaction(t2, C2)
Tile2->InsertTransaction(t2, C3)
Tile0
Tile1
Tile2

Figure 1: Inter-tile Communcation.

```
};
```

C.5.1 Transactions:
All communication between tiles is done through Transactions. We provide a generic transaction base class. Tile designers can then inherit that class to add special fields required for the communication and synchronization needs of their tiles. This could also include details like data width and special transaction types that a tile uses to determine the required actions (e.g., forward a message to an intermediary tile, send a direct response, etc.) or number of cycles it would take to process a request. Above is the code for the the Transaction base class, from pythia/sim/tile/Tile.h.

C.6 Putting it Together

C.6.1 Instantiating a Tile:

1. Design a tile subclass that inherits the tile class in pythia/sim/tile/Tile.h and place the subclass source file in a new pythia/sim/tile/chosen_name directory.
2. In pythia/sim/tile/Tile.h, create new Transaction types and subclasses relevant for the communication of the tiles.
3. Code your main file and place it in pythia/sim/tile/chosen_name/main.cc, matching the code snippet below:

```
#include "../../sim.h" //include the header file for the simulator
#include "tile/chosen_name/ExampleTile.h" //include header file for your tile

int main(int argc, char const *argv[]) {
    Simulator* simulator=new Simulator();
    Tile* tile0 = new ExampleTile(simulator,2000);
    Tile* tile1 = new ExampleTile(simulator,333);
    simulator->registerTile(tile0,0); //assign tile id 0 to tile0
    simulator->registerTile(tile1,1); //assign tile id 1 to tile1

    simulator->run(); //run the simulator, which steps through each of the tiles until they're
        done working
    return 0;
}
```

4. In pythia/sim/CMakeLists.txt, remove "main.cc" and add "tile/chosen_name/main.cc" and "tile/chosen_name/ExampleTile.cc" to the list of source files in "ADD_EXECUTABLE".
5. Compile the simulator as normal.
6. Cd to pythia/bin and run the simulator with ./sim (add any command line arguments as required by your main file)

C.6.2 Integrating Tiles with Default Pythia Cores: Pythia comes with default core tile models, each with configurable hardware structures, a private L1 cache, and a shared L2 cache. We anticipate that it would be useful to integrate new tiles with the existing core tiles. Below are the steps to design and instantiate a tile with the default core tiles.

1. Follow steps 1-2 in Section C.6.1 above.
2. In pythia/sim/main.cc, find the commented region (shown in the code snippet below) and, just above it, create a new tile and register it, as shown below.

```
//snippet from pythia/sim/main.cc

/********
register the other tiles here
e.g.
Tile* tile = new ExampleTile(simulator,2000);
simulator->registerTile(tile,tileid); //manual tile id assignment
or
simulator->registerTile(tile); //auto tile id assignment
********/
```

3. In pythia/sim/CMakeLists.txt, add "tile/ExampleTile.cc" to the list of source files in "ADD_EXECUTABLE".
4. Compile and run the simulator as normal!

D Demos

In this section, we document our video demonstrations of our cross-stack DECADES toolflow, from applications to compiler to simulator and hardware targets.

D.1 Applications

The videos below demonstrate the DECADES architecture and compiler innovations that have yielded significant speedups SDH-assigned applications.

D.1.1 Graph projections: A graph algorithm written in C++: `https://youtu.be/vGHEbp-VaIc`

D.1.2 Vertex Nomination: A graph algorithm written in Python: `https://youtu.be/cilW-gVaMsc`

D.2 Compiler

The video below highlights the ability to use DEC++ to generate RISC-V binaries that can run on the DECADES FPGA emulation and chip platforms.
`https://youtu.be/sqsImYyPgBg`

D.3 Hardware

Below is a video demonstration of the hardware component of the full-stack DECADES toolflow. It highlights the ability to run RISC-V binaries created by DEC++ as part of the DECADES FPGA emulation and chip platforms
`https://youtu.be/wMi8XkxWQ9E`

D.4 Simulator

Below, we demonstrate compiling with DEC++ and running applications with our simulator Pythia. The demo showcases two features: do-all parallelism and decoupling.
`https://youtu.be/hUThWUhkEWg`

E Appendix: Original proposal

A. Innovative Claims: Modern compute systems use increasing amounts of heterogeneous and accelerator-oriented parallelism to meet aggressive performance/power targets. As accelerators have sped up application *compute* portions, the Amdahl's bottleneck in these applications has become the *data supply problem*. As a relative fraction of runtime, the key bottlenecks lie in memory and communication overheads associated with supplying these accelerators with data. With both TA-1 and TA-2 efforts, DECADES will develop (i) reconfigurable hardware platforms offering flexible specialization to accelerate data supply by 2-3 orders of magnitude for the targeted workloads, and (ii) software toolchains statically and dynamically optimizing data supply.

Our prior work on Graphicionado [6] and [5] offered two orders of magnitude improvements in performance-per-watt over best-in-class software approaches, by attacking data supply issues. Key strategies are (1) compiler and runtime techniques for *decoupling* memory accesses to provide lookahead and prefetch relative to the calculations using that data, and (2) hardware support including both novel storage and novel memory instruction retirement to facilitate that lookahead.

Our innovative claims are: (1) **Tailoring for locality:** LLVM-based compiler optimizations to enhance locality, with scratchpads, queues, and local storage custom-mapped on chip per application access patterns. (2) **Decoupling data supply from compute:** Compiler slicing and reference granularity optimizations to improve lookahead and latency tolerance by decoupling data supply and memory address calculations from compute itself; hardware support for decoupling and lookahead, including terminal load optimizations. (3) **Reconfigurable hardware platform:** Heterogeneous tile-based architecture with three main classes of reconfigurable tiles (processors, accelerators, and storage) connected via a reconfigurable network with multiple DRAM partitions, each equipped with reconfigurable in-memory data-management logic. (4) **Scalable simulation and FPGA emulation:** Innovative support for design and evaluation of heterogeneous hardware-software systems. (5) **Full chip design:** Demonstrate system improving 500X-1000X GOPS/Watt over SDH-selected reference processor.

B. Technical Approach: Fig. 2 illustrates the distinctive properties and inter-dependencies of the proposed TA-1 and TA-2 efforts. To achieve the TA-1 goals, we will develop a *reconfigurable hardware platform* that consists of a main chip that communicates through multiple high-speed links with a group of off-chip DRAM subsystems, which are also inter-connected through a dedicated ring. The *DECADES chip* is based on a heterogeneous tile-based architecture where three types of reconfigurable tiles (processor, accelerator and intelligent storage) are connected through a highly *reconfigurable network-on-chip (NoC)*. Each DRAM subsystem is accessed through an FPGA providing fast reconfigurable functions for in-memory data management via pre-designed bitstreams.

To achieve the TA-2 goals, we will develop a *compiler* and a *runtime system* that are optimized to solve the data supply problem for data-intensive workloads in the target domains of dense linear algebra and massive graph analytics. They will process the target applications to identify two types of logical building blocks: *CompD tasks*, which perform the main problem calculations, and *SuppD tasks*, which perform data supply for CompDs. Based on its characteristics, each task will be mapped onto either a processor or an accelerator tile. Each processor tile contains a set of *configuration knobs* that can be dynamically set to optimize the execution of the given CompD or SuppD task, e.g. by activating/disabling some functional units or by determining a given degree of instruction-level parallelism. Similarly, each particular accelerator tile contains a specific set of configuration knobs, e.g. to optimize the precision of its datapath arithmetic for a give CompD task or to adapt part of its datapath logic to support some specific memory addressing for a given SuppD task. In parallel, the DECADES compiler and runtime system can reconfigure the rest of the DECADES hardware platform to best support the current workloads with various optimization, including for example: (i) dynamically reconfigure each storage tile to operate in one of various different modalities (caches, scratchpads, queues...), (ii) reconfigure portions of the NoC to best support the current mapping of CompD and SuppD across the tiles, and (iii) configure the in-memory logic within the DRAM subsystems to move data across banks through the external ring.

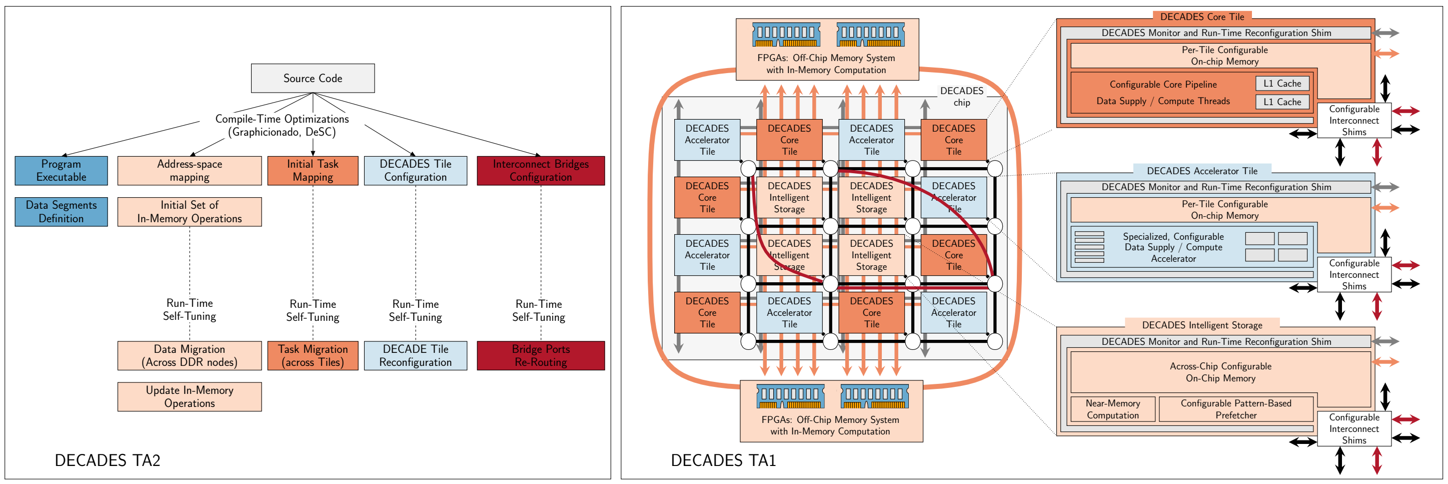
Source Code
Compile-Time Optimizations (Graphicionado, DeSC)
Program Executable
Data Segments Definition
Address-space mapping
Initial Set of In-Memory Operations
Initial Task Mapping
DECADES Tile Configuration
Interconnect Bridges Configuration
Run-Time Self-Tuning
Run-Time Self-Tuning
Run-Time Self-Tuning
Run-Time Self-Tuning
Data Migration (Across DDR nodes)
Update In-Memory Operations
Task Migration (across Tiles)
DECADE Tile Reconfiguration
Bridge Ports Re-Routing
DECADES TA2
FPGAs: Off-Chip Memory System with In-Memory Computation
DECADES chip
DECADES Accelerator Tile
DECADES Core Tile
DECADES Intelligent Storage
FPGAs: Off-Chip Memory System with In-Memory Computation
DECADES Core Tile
DECADES Monitor and Run-Time Reconfiguration Shim
Per-Tile Configurable On-chip Memory
Configurable Core Pipeline
Data Supply / Compute Threads
L1 Cache
L1 Cache
Configurable Interconnect Shims
DECADES Accelerator Tile
DECADES Monitor and Run-Time Reconfiguration Shim
Per-Tile Configurable On-chip Memory
Specialized, Configurable Data Supply / Compute Accelerator
Configurable Interconnect Shims
DECADES Intelligent Storage
DECADES Monitor and Run-Time Reconfiguration Shim
Across-Chip Configurable On-Chip Memory
Near-Memory Computation
Configurable Pattern-Based Prefetcher
Configurable Interconnect Shims
DECADES TA1

Figure 2: DECADES will use aggressive, application-specific compile-time, pre-run, and run-time analysis and adaptation to tightly tailor memory hierarchies to particular software/input tasks. Our work will demonstrate our approach first in simulations (Phase 1) and then in FPGA emulation (Phase 2) and a culminating chip design (Phase 3)

B.1 Data Supply Analysis and Optimization: DECADES compiler and runtime optimizations will use *compiler slicing* to increase lookahead by decoupling memory accesses and related address calculations from the subsequent computations that use this data. Unlike prefetching, we perform this lookahead as true data supply, so that the subsequent compute task (CompD) can be a pure accelerator with no connection to memory. This saves off-chip memory bandwidth by setting up SuppD-CompD and CompD-CompD data handoffs, rather than communication through the memory system. Our LLVM-based compiler will optimize the mapping of SuppDs and CompDs and configure the DECADE platform to form *computational pipelines for communication locality*.

SuppDs can be greatly optimized by exploiting (hardware) support for *multi-granular or vectorized data supply instructions*. Such instructions allow for choice between typical (e.g. word) granularity references and very coarse-grained references but at much lower performance/energy overhead than DMA. Our compiler will *mitigate common off-chip memory bottlenecks* by synthesizing and mapping SuppDs whose sole purpose is to optimize the on-chip flow of data. With coarse-grained memory accesses, this first-line SuppD can perform page-granularity fetches and then re-tailor the data for better locality before feeding it out to individual CompDs.

Dynamic Tailoring can optimize further when application data set size and locality characteristics are not known until runtime. For example, the page-granularity SuppD previously mentioned is only beneficial if enough data on each page will be accessed. Runtime monitoring will guide characterizations of the types and granularities of optimizations expected to pay off. If insufficient locality (data usage) is observed, dynamic reconfiguration can shift towards smaller fetch granularities. Other dynamical tailoring will *adjust the degree of lookahead or decoupling employed*. Aggressive lookahead helps the system improve memory latency tolerance, but can require larger on-chip scratchpads and queues which cost energy. While much of our decoupling is non-speculative, additional lookahead can be gained by using control or memory speculation, but with an energy cost if it proves incorrect and requires rollback. Our work will adjust both non-speculative and speculative lookahead amounts based on measured performance and power targets.

B.2 Reconfigurable Hardware Platform: As shown in Fig. 2, the processor, accelerator and storage elements are tiled across the DECADES chip. A *reconfigurable NoC* supports tailored communications connectivity among the tiles in order to best support the data flows identified during compilation and runtime monitoring. *Runtime self-tuning* extends to the main memory subsystems, where configurable in-memory operations support direct data migration across the DRAM banks via the external ring, without involving the chip. All tiles are configurable to be tailored to workloads at compile time, and further runtime reconfiguration is also available. For example, a processor tile can be customized to produce data in the order and granularity that matches the needs of a given CompD task. The degree of memory lookahead can be configured, as can be the granularity at which it is fetched. Configurations can either be fully static, or offer runtime-adjustable choices (e.g. multiple fetch granularities).

To support aggressive decoupling between SuppD and CompD, we will implement *terminal load optimization* as in [5]. Once compiler slicing has separated memory address calculations (SuppD) from problem calculations (CompD), the SuppD slice typically has many newly-identified *terminal loads* which have no further usage on the SuppD (only on CompD). Optimizing the retirement of these often-high-latency memory operations offers extensive lookahead at low hardware cost. Our TA2 compiler algorithms will identify and exploit terminal loads.

Runtime Monitoring and *Interface Shims* will support adjustment of parameter choices. Processor and accelerator tiles can be configured to select performance counters for metrics, such as assessing SuppD lookahead. Insufficient SuppD-CompD lookahead can be obtained by measuring miss rates on associated caches, or wait times on associated scratchpads or queues. Performance counters at interface shims also measure metrics on the traffic flow and degree of contention at different NoC points.

B.3 Scalable Full-System Simulation and Emulation of DECADES Systems: We will first develop a *lightweight simulation engine* to evaluate early-stage tradeoffs in the design of the reconfigurable hardware platform as as well as to test the compiler optimizations. This simulator will be based on PIN and PriME [4] and optimized to model different types of processors and accelerators as well as the reconfigurable storage and interconnect. We will then transition the viable portions of this simulator into a *full-system simulator* which will be capable of capturing the complex communication and memory-access patterns that can happen in large multicore architectures executing many concurrent workloads on top of a complete software stack (operating system, drivers, and middleware). This is necessary to validate the proposed hardware shims and compiler techniques for communication optimization.

Our full-system simulator will be structured in two main parts: an *emulation front-end* and a *timing/energy modeling back-end*. To simulate the processors, we will use QEMU, a highly portable virtual platform that supports two dozen different guest ISAs and eight host ISAs. To simulate the accelerators, the reconfigurable storage and the NoC, we will use the IEEE-standard SystemC language, by instancing one SystemC simulator engine in a separate thread for each accelerator. Hardware implementations can be automatically synthesized from SystemC specifications with *high-level synthesis (HLS)* tools. This allows us to use the same SystemC code both in our simulator and to synthesize actual hardware for FPGA emulation and even for the DECADES chip.

To develop a scalable full-system simulator for heterogeneous architectures with the right 'speed vs. accuracy' balance, we will leverage our experience in realizing a scalable dynamic binary translator that is simple, memory-efficient and correct [3]. We contributed this translator to the upstream distribution of QEMU; its speed scales with host system core count comparably to native execution and hardware-assisted virtualization. We will also leverage the experience of PI Carloni's group in designing many complex accelerators with SystemC and HLS.

B.4 Chip Design: A critical aspect of designing novel software/hardware systems is prototyping those complete systems through full hardware implementations. We will realize a DECADES chip prototype in order to truly understand the hardware implementation costs in terms of area, energy, and cycle time and to provide an at-speed and at-scale platform to execute SDH software. By creating a chip-level prototype, the DECADES approach is significantly de-risked for transition to deployed military and commercial use.

The chip will implement a complete instance of the DECADES reconfigurable hardware platform in an advanced technology node (16nm, 14nm, or 7nm) leveraging DARPA/MTO shuttle runs. The chip will include a large number of processor and accelerator tiles together with customizable storage and configurable interconnect. Significant design will go into interfacing with off-chip DRAM and high-speed I/O as both are required to feed the DECADES chip. We will build a custom printed circuit board to host the chip, the DRAMs, and the FPGAs providing in-memory computing capabilities. Significant effort will be expended to make both the chip and board the ideal evaluation mechanism for energy and performance. For instance, all power rails will be isolated on the board in order to enable individual characterization of the different components. Power and on-chip voltage monitors will be included in the design. A key research aspect is the integration of the chip into a complex software stack with a full blown operating system (OS). In order to run an OS like Linux, we will use an open source processor core to make the DECADES chip completely self hosting and to study the OS-to-accelerator interfacing issues. Also by including an open source processor, we can demonstrate DECADES's 500x-1000x GOPS/Watt improvement implemented in the same process technology, enabling an apples-to-apples comparison point.

In implementing the DECADES chip, we will draw on our team's proven track record of building academic and commercial processors including Raw, Tilera and Piton. We will also leverage the results of two projects supported by the DARPA PERFECT program: the open source OpenPiton processor [1] and the ESP system-level design methodology and accelerator library [2].

C. Deliverables: See Table 1. The deliverables offer highest leverage when TA1 and TA2 work together, but can also be demonstrated separately. There are no proprietary claims.

D. Other Research: Static optimization and dynamic adaptation proposals relate to automatic parallelization research and CPU-GPU communication optimizations such as DSWP; our work goes further in assuming reconfigurable components with high-leverage hardware support. Execution-based prefetching techniques construct a thread by hand, compiler, or hardware and then run this thread to prefetch. In runahead execution the processor continues through cache misses to speculatively prefetch data. DECADES is targeted for decoupled compute accelerators that may have no memory connection and where speculation is unsuitable. Prior parallel configurable heterogeneous architectures include Widget and Plasticine. DECADES is more configurable than them, and more energy- and area-efficient due to its static/dynamic mix. Many other within-CPU latency tolerance techniques have been proposed (e.g., the KiloInstruction Processor). DECADES' distinct SuppD and CompD slices greatly increase the potential value of terminal load and other optimizations. We also note prior hardware specializations for particular application domains [7, 6]; DECADES is more generalizable.

E. Schedule:

Table 1: Project schedule and deliverables.

	Phase 1	**Phase 2**	**Phase 3**
Compiler Analysis (TA2)	Locality and Decoupling; Multigranular and Vectorized References	SuppD Synthesis and Mapping	–
Runtime Adaptation (TA1)	Reconfigurable Hardware Platform Architecture	Adaptive queue and scratchpad sizing	Decoupling and speculation adaptations
Simulation & Emulation (TA1 & TA2)	Early-stage simulation engine based on PIN and PriME [4].	FPGA Emulation seeded from simulation results	FPGA Emulation with dynamic code adaptations
Chip Design (TA1)	Collect relevant RTL modules, templates, physical design IP	FPGA prototyping to guide chip design; initial chip parameter studies	Chip design, tapeout, bringup, characterization
Deliverables	**TA1:** Derive complete specification of proposed DECADES hardware and configuration options. Develop simulation environment supporting TA2 development and testing. **TA2:** Develop compiler flow for data supply optimization by i) decoupling memory accesses from their compute use, ii) selecting best SuppD, CompD mappings, along with storage and interconnect.	**TA1:** Complete initial DECADES hardware design and prototype it as a fully-functional configurable engine using FPGAs. **TA2:** Develop adaptive storage optimizations (queue and scratchpad size, lookahead) based on measured program characteristics.	**TA1:** Refine DECADES architecture; Complete chip implementation. Develop test board with proposed chip, memory, test functionality. We will make use of DARPA/MTO fab shuttle runs (USG furnished) for these efforts. **TA2:** Integrate toolchain's static, dynamic optimizations. Evaluate full HW-SW system by executing target D3M/HIVE apps compiled with TA2 toolchain on TA1 hardware.

References

[1] J. Balkind et al. OpenPiton: An open source manycore research framework. In *ASPLOS*, 2016. *Nominated for Best Paper*.

[2] L. P. Carloni. The case for embedded scalable platforms. In *Proc. of the Design Automation Conf. (DAC)*, June 2016.

[3] E. G. Cota, P. Bonzini, A. Bennee, and L. P. Carloni. Cross-ISA machine emulation for multicores. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, February 2017.

[4] Y. Fu and D. Wentzlaff. Prime: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[5] T. J. Ham, J. L. Aragon, and M. Martonosi. DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. In *MICRO-48*, 2015.

[6] T. J. Ham et al. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO-49*, 2016. *Best Paper Award*.

[7] M. M. Ozdal et al. Energy efficient architecture for graph analytics accelerators. In *ISCA*, 2016.