A Universal Parallel Front End for Execution Driven Microarchitecture Simulation

Chad D. Kersey
Sudhakar Yalamanchili
Georgia Institute of Technology

Arun Rodrigues
Sandia National Laboratories

Outline

Outline

Outline

Outline

- Introduction
  - Simulators
  - Front Ends

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End
  - API Overview

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End
  - API Overview
  - How is QSim “Universal”?

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End
  - API Overview
  - How is QSim "Universal"?
  - How is it Parallel?

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End
  - API Overview
  - How is QSim "Universal"?
  - How is it Parallel?
  - How does it Perform?

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End
  - API Overview
  - How is QSim "Universal"?
  - How is it Parallel?
  - How does it Perform?
  - How are QEMU and QSim Related?

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End
  - API Overview
  - How is QSim "Universal"?
  - How is it Parallel?
  - How does it Perform?
  - How are QEMU and QSim Related?
- Back Ends

Outline

- Introduction
  - Simulators
  - Front Ends
- The QSim Simulator Front End
  - API Overview
  - How is QSim "Universal"?
  - How is it Parallel?
  - How does it Perform?
  - How are QEMU and QSim Related?
- Back Ends
- Summary

Outline

Introduction

Introduction– The Ubiquity of Simulation

- Simulation is a requirement of architecture research.

Introduction– The Ubiquity of Simulation

- Simulation is a requirement of architecture research.
- Few architecture researchers have access to the resources needed to create full-scale prototypes.

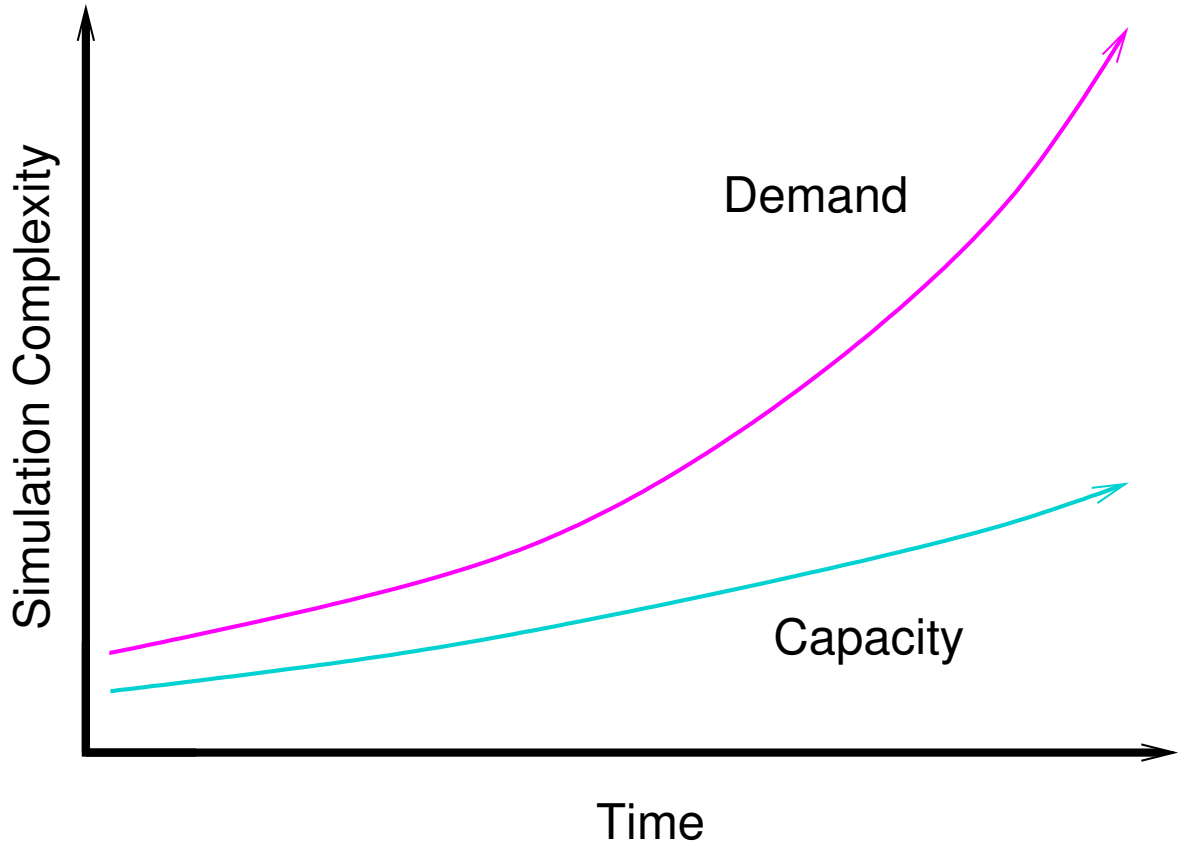
Introduction– The Ubiquity of Simulation

- Simulation is a requirement of architecture research.
- Few architecture researchers have access to the resources needed to create full-scale prototypes.
- Those with the resources would prefer not to spend them building incremental prototypes.

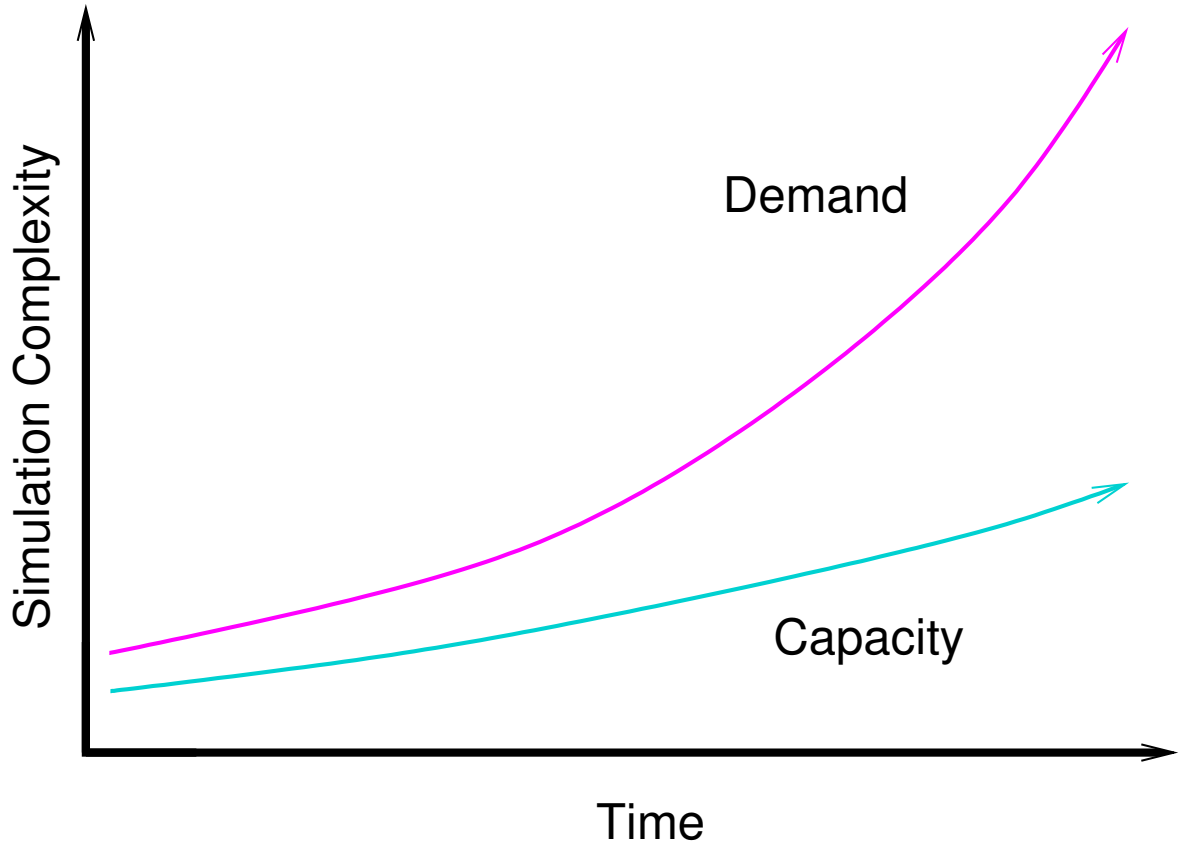
Introduction– The Ubiquity of Simulation

- Simulation is a requirement of architecture research.
- Few architecture researchers have access to the resources needed to create full-scale prototypes.
- Those with the resources would prefer not to spend them building incremental prototypes.
- Even if they would, the turn-around time for building a new CPU, even using pre-designed components would be very long.

Introduction– The Simulation Gap

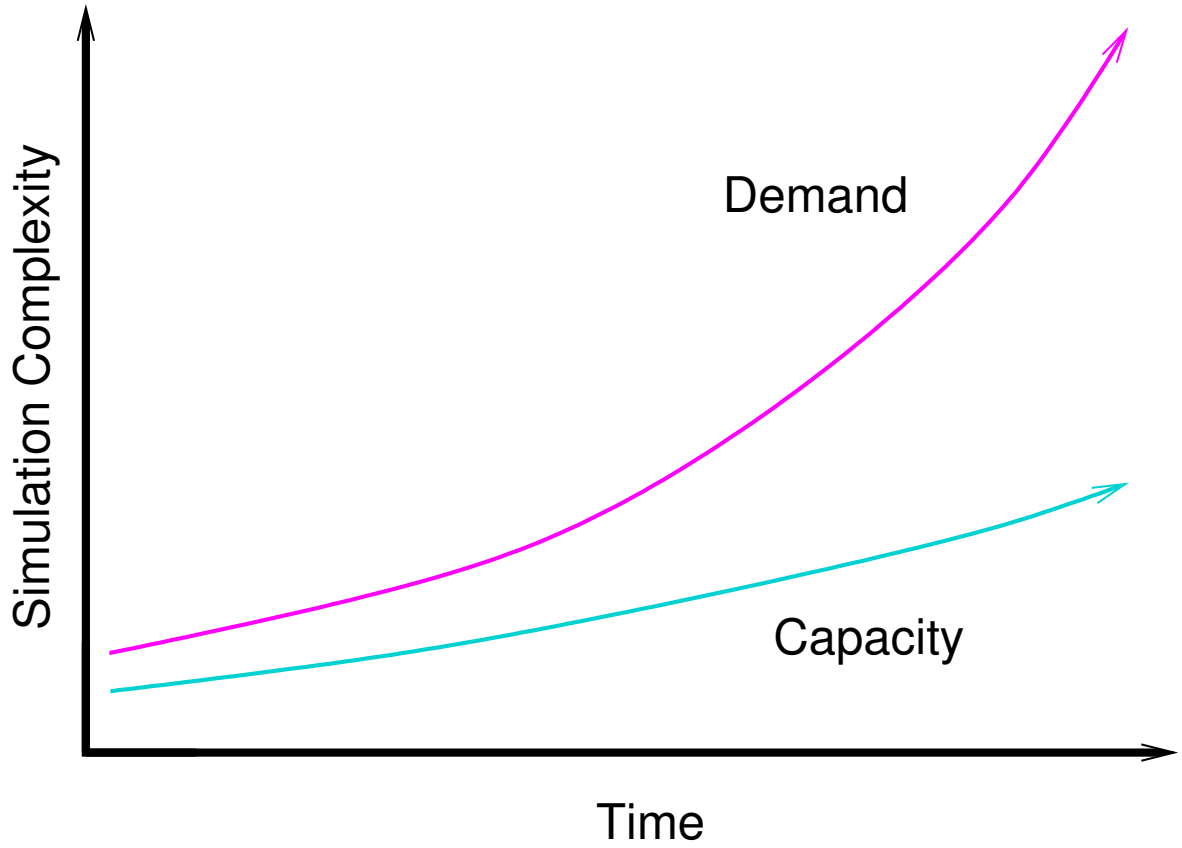


Introduction– The Simulation Gap



Reasons for the simulation gap:

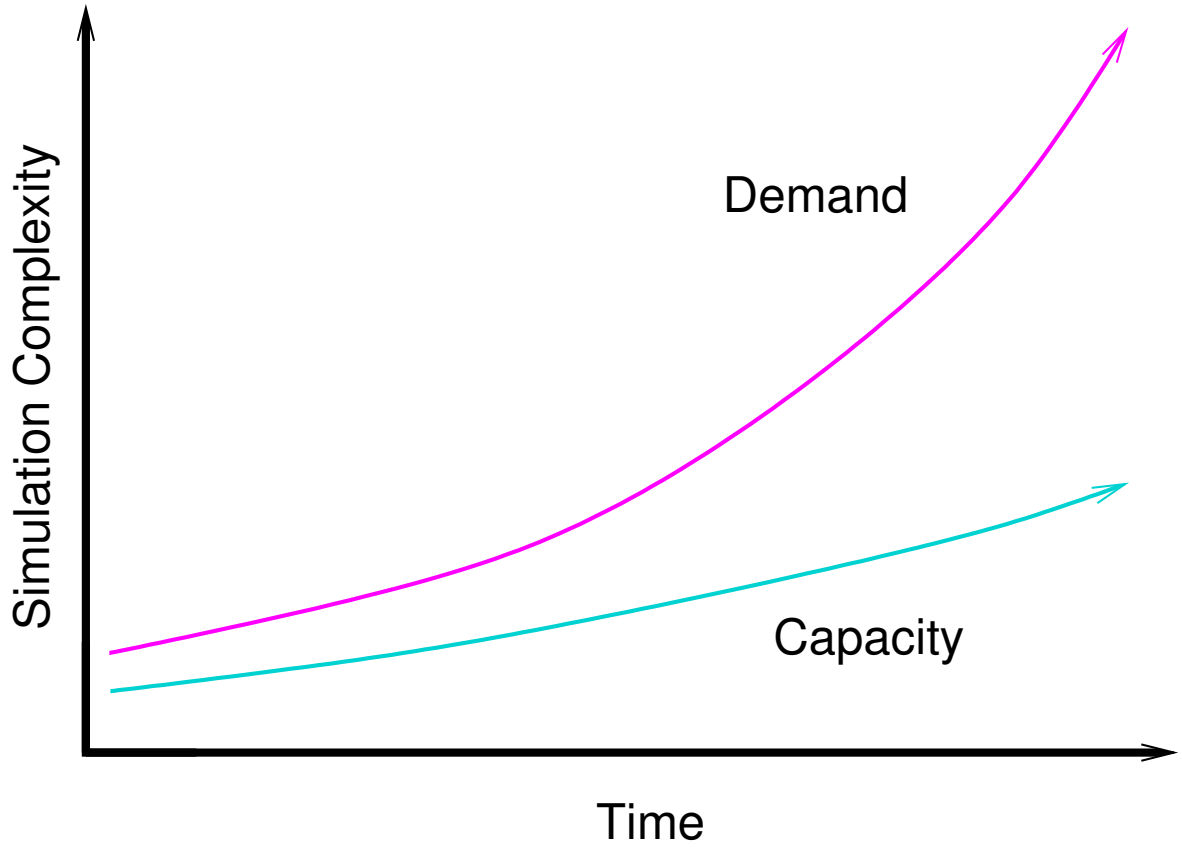
Introduction– The Simulation Gap



Reasons for the simulation gap:

- Parallel simulation is hard, so we use serial simulators for parallel machines.

Introduction– The Simulation Gap



Reasons for the simulation gap:

- Parallel simulation is hard, so we use serial simulators for parallel machines.
- Developments in computer architecture tend to be additive, but we keep building simulators from scratch.

Introduction– Narrowing the Gap

Ways to narrow the simulation gap:

Introduction– Narrowing the Gap

Ways to narrow the simulation gap:

- Spend less time researching architecture and more time developing simulators?

Introduction– Narrowing the Gap

Ways to narrow the simulation gap:

- Spend less time researching architecture and more time developing simulators?
  - Probably would not be well-received by the architecture community.

Introduction– Narrowing the Gap

Ways to narrow the simulation gap:

- Spend less time researching architecture and more time developing simulators?
  - Probably would not be well-received by the architecture community.
- Increase simulator throughput so more simulations can be run in a reasonable amount of time.

Introduction– Narrowing the Gap

Ways to narrow the simulation gap:

- Spend less time researching architecture and more time developing simulators?
  - Probably would not be well-received by the architecture community.
- Increase simulator throughput so more simulations can be run in a reasonable amount of time.
  - Parallelize them.

Introduction– Narrowing the Gap

Ways to narrow the simulation gap:

- Spend less time researching architecture and more time developing simulators?
  - Probably would not be well-received by the architecture community.
- Increase simulator throughput so more simulations can be run in a reasonable amount of time.
  - Parallelize them.
- Find ways to make simulator development more efficient.

Introduction– Narrowing the Gap

Ways to narrow the simulation gap:

- Spend less time researching architecture and more time developing simulators?
  - Probably would not be well-received by the architecture community.
- Increase simulator throughput so more simulations can be run in a reasonable amount of time.
  - Parallelize them.
- Find ways to make simulator development more efficient.

If we make simulator development more efficient, we increase the rate at which simulation capacity can grow.

Introduction– Front Ends

What is a front end?

Introduction– Front Ends

What is a front end?

- Most simulators are broken into a *front end* and a *back end* by their designers.

Introduction– Front Ends

What is a front end?

- Most simulators are broken into a *front end* and a *back end* by their designers.
- The front end handles the execution of instructions (making sure the register state is correct).

Introduction– Front Ends

What is a front end?

- Most simulators are broken into a *front end* and a *back end* by their designers.
- The front end handles the execution of instructions (making sure the register state is correct).
- Because instruction sets are very complex, front ends are usually created by using and modifying an existing emulation solution or avoiding emulation entirely and tracing native execution.

Introduction– Front Ends

What is a front end?

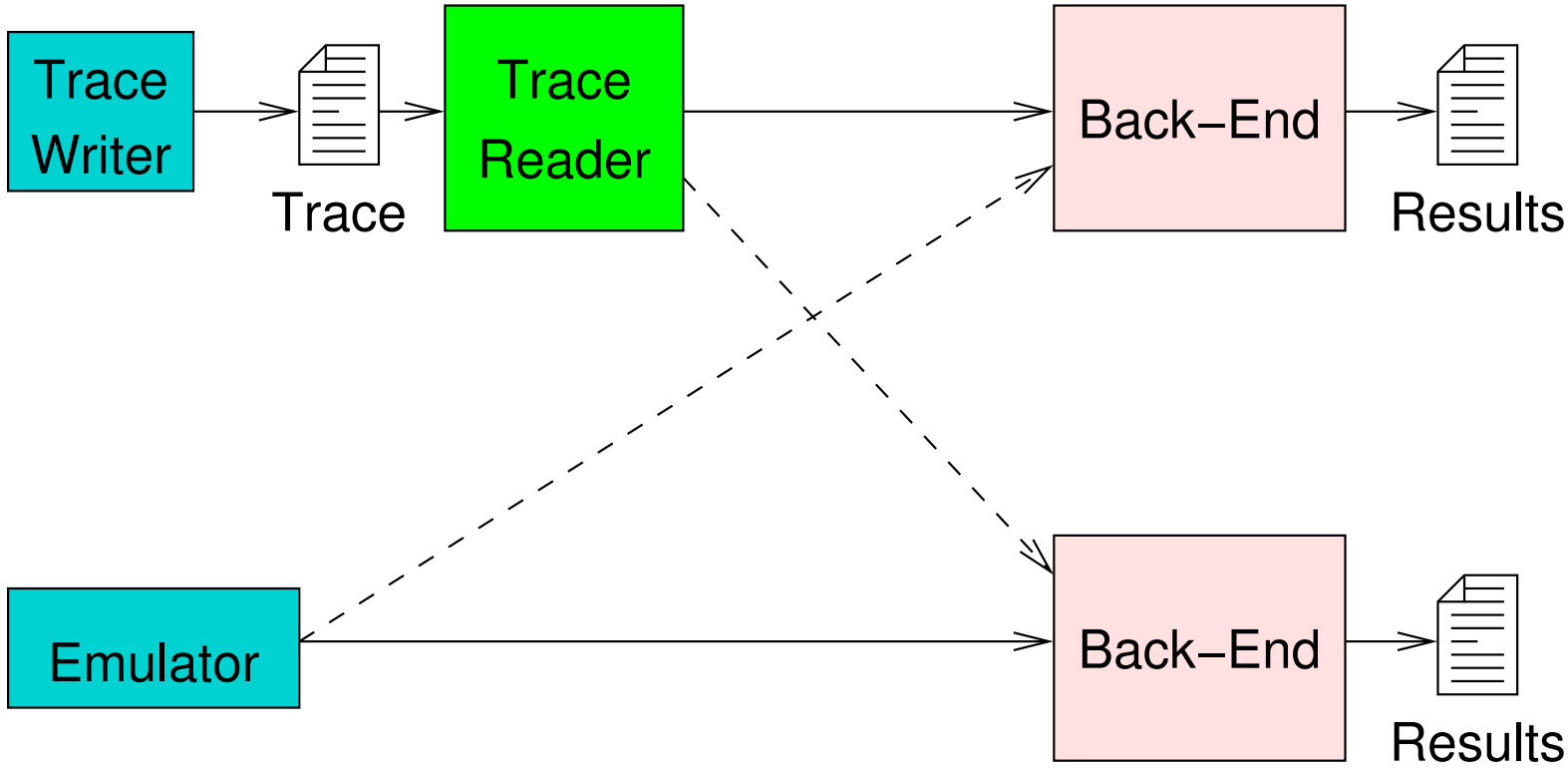
- Most simulators are broken into a *front end* and a *back end* by their designers.
- The front end handles the execution of instructions (making sure the register state is correct).
- Because instruction sets are very complex, front ends are usually created by using and modifying an existing emulation solution or avoiding emulation entirely and tracing native execution.
- The back end handles timing, power, and other metrics (how long did that instruction take to clear the pipeline).

Introduction– Front Ends

What is a front end?

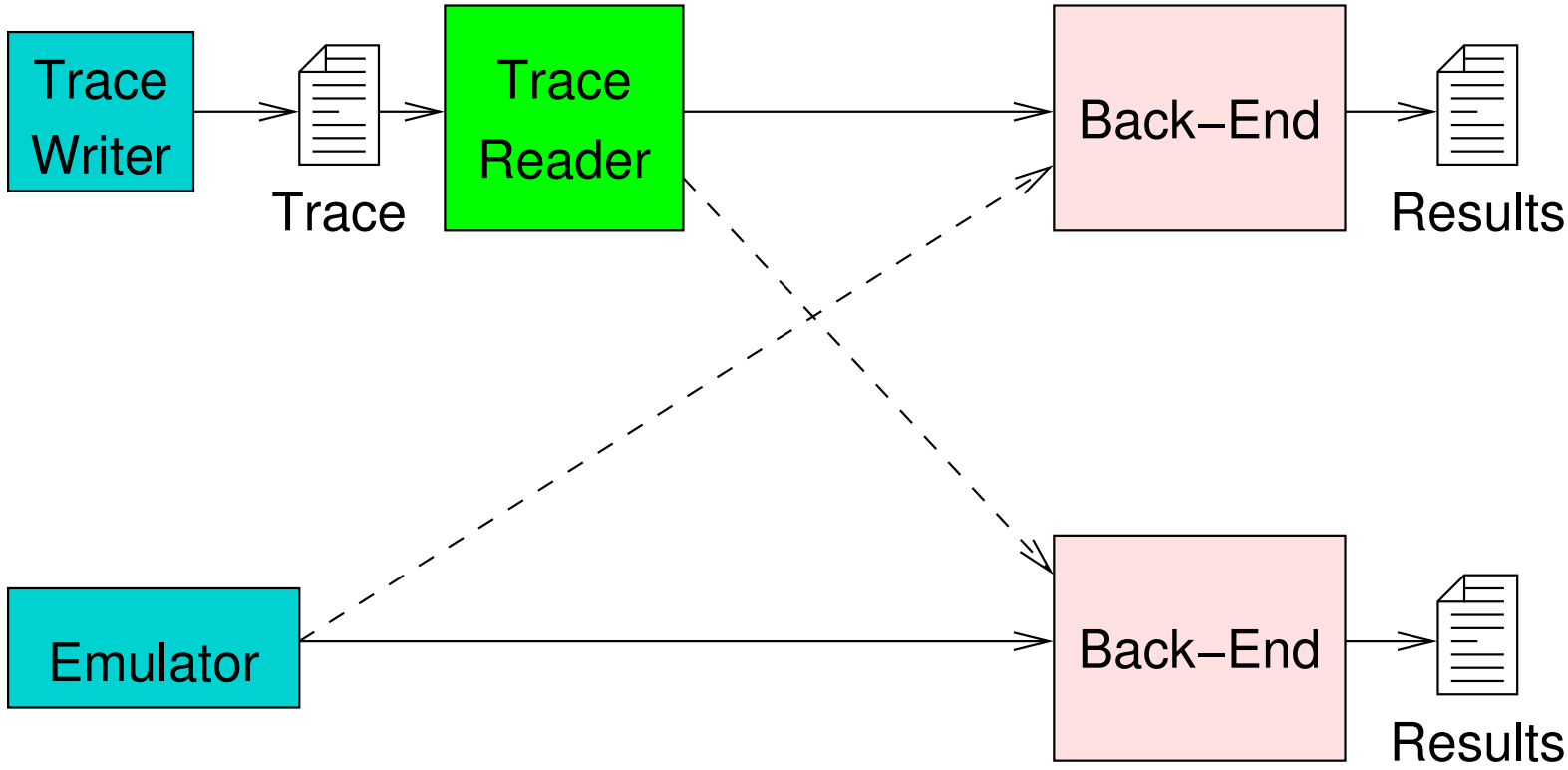
- Most simulators are broken into a *front end* and a *back end* by their designers.
- The front end handles the execution of instructions (making sure the register state is correct).
- Because instruction sets are very complex, front ends are usually created by using and modifying an existing emulation solution or avoiding emulation entirely and tracing native execution.
- The back end handles timing, power, and other metrics (how long did that instruction take to clear the pipeline).
- Back ends are the part that implements the logic that makes a simulator unique.

Introduction– The Ideal Front End



Ideally:

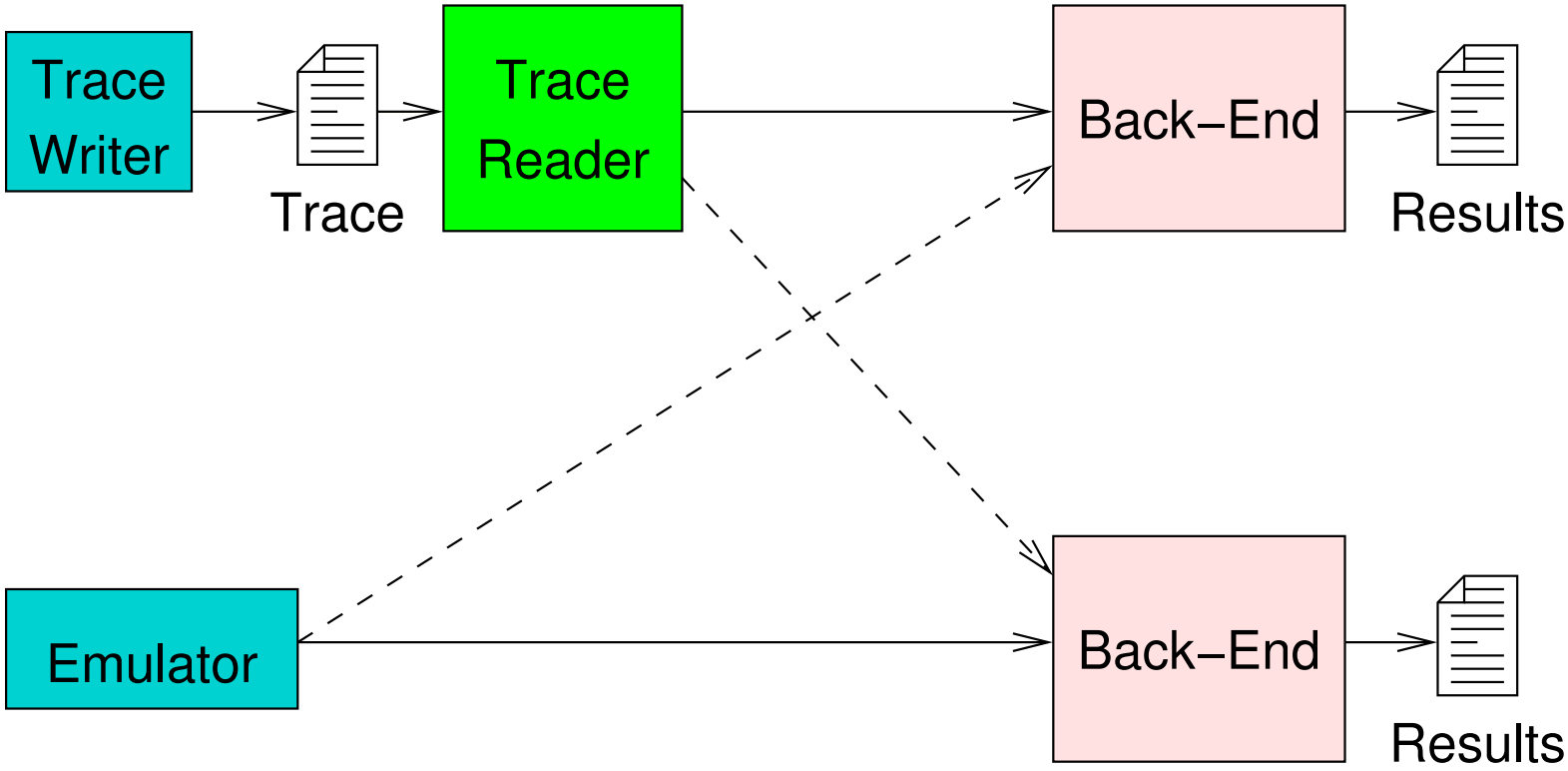
Introduction– The Ideal Front End



Ideally:

- Each front end and back end must be written only once, after which they can be used in any combination, like compier front ends and back ends.

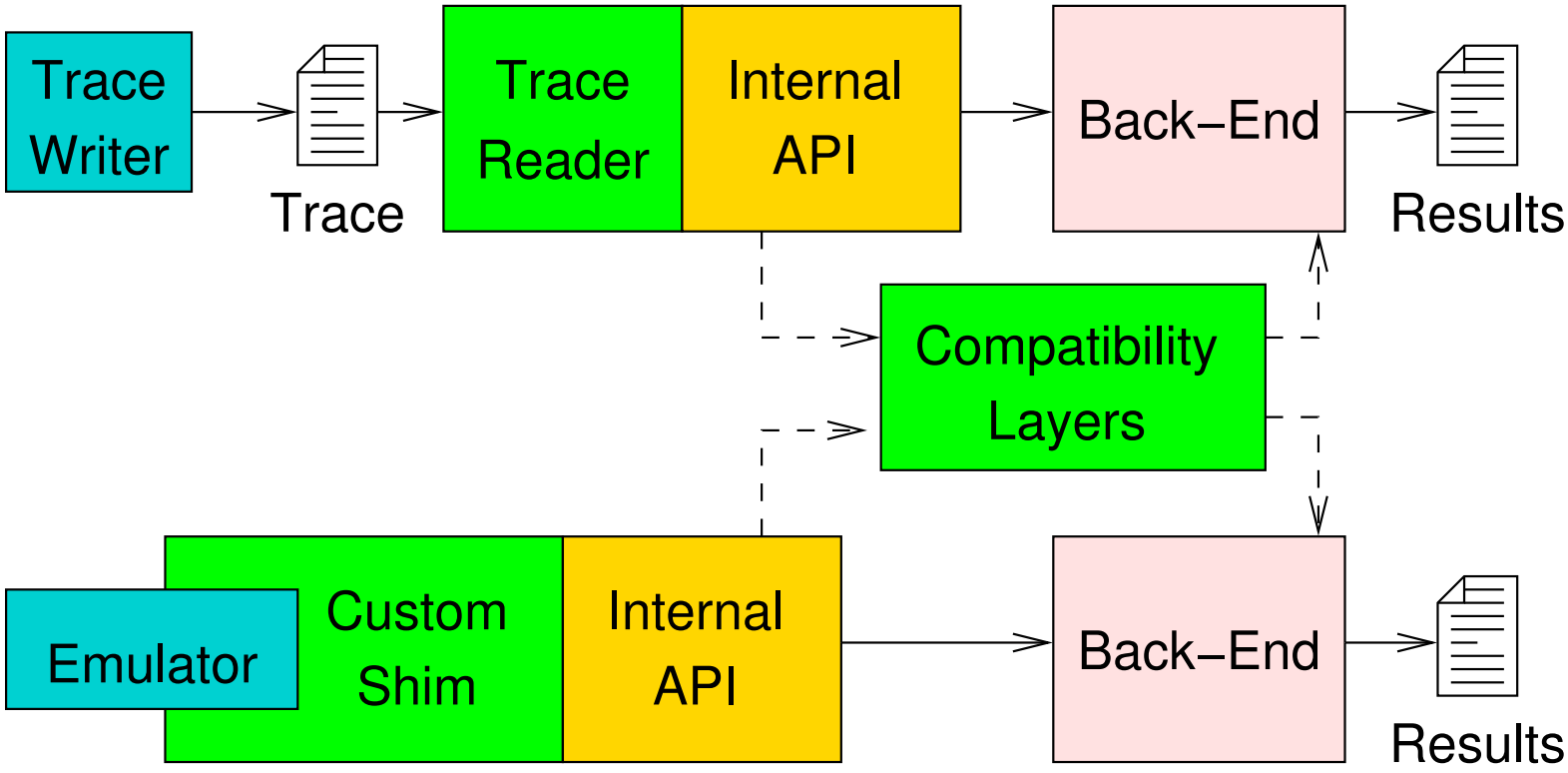
Introduction– The Ideal Front End



Ideally:

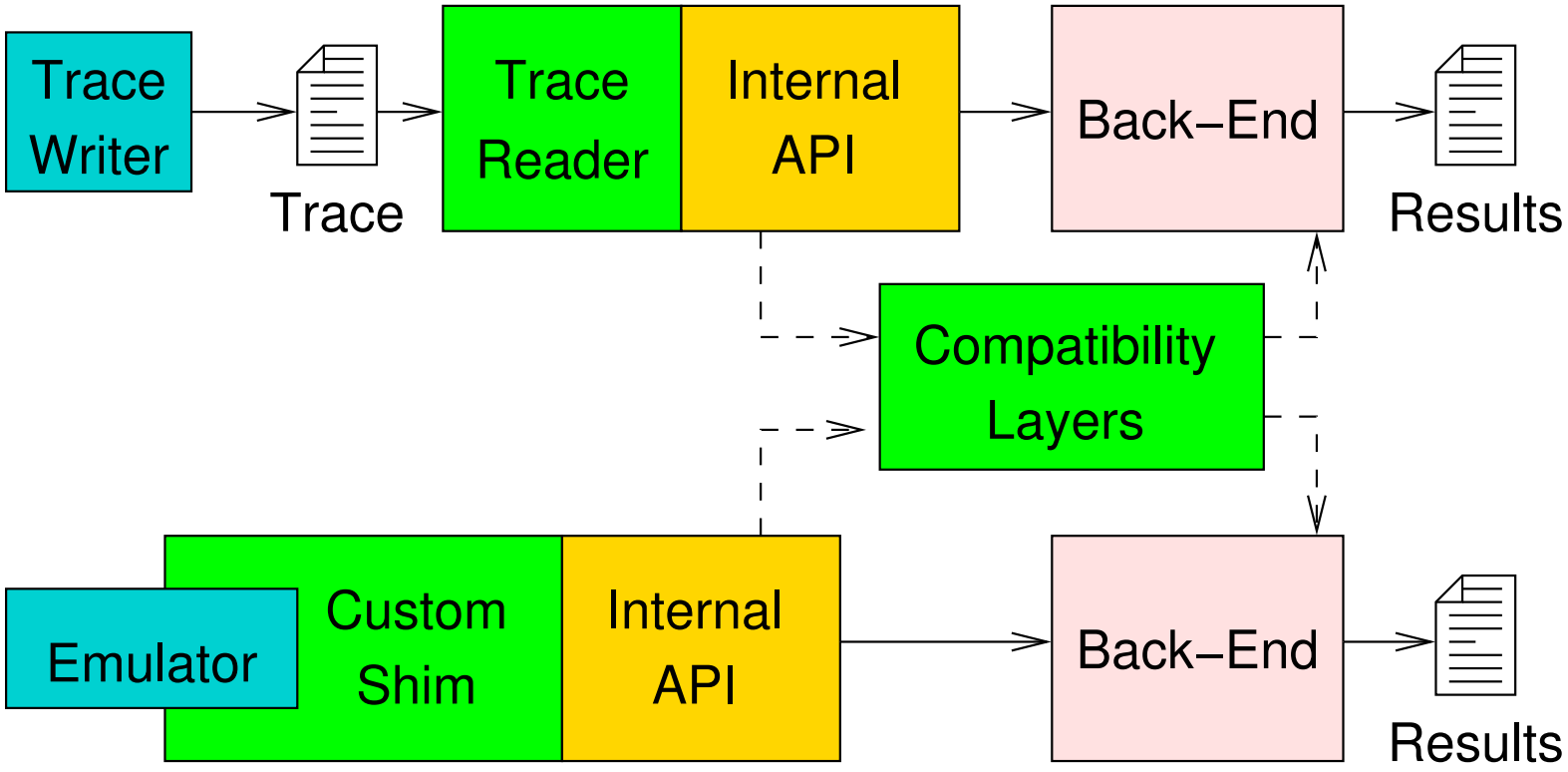
- Each front end and back end must be written only once, after which they can be used in any combination, like compier front ends and back ends.
- No additional code would need to be written to adapt general purpose emulators for simulation duty.

Introduction– Real Front Ends



Realistically:

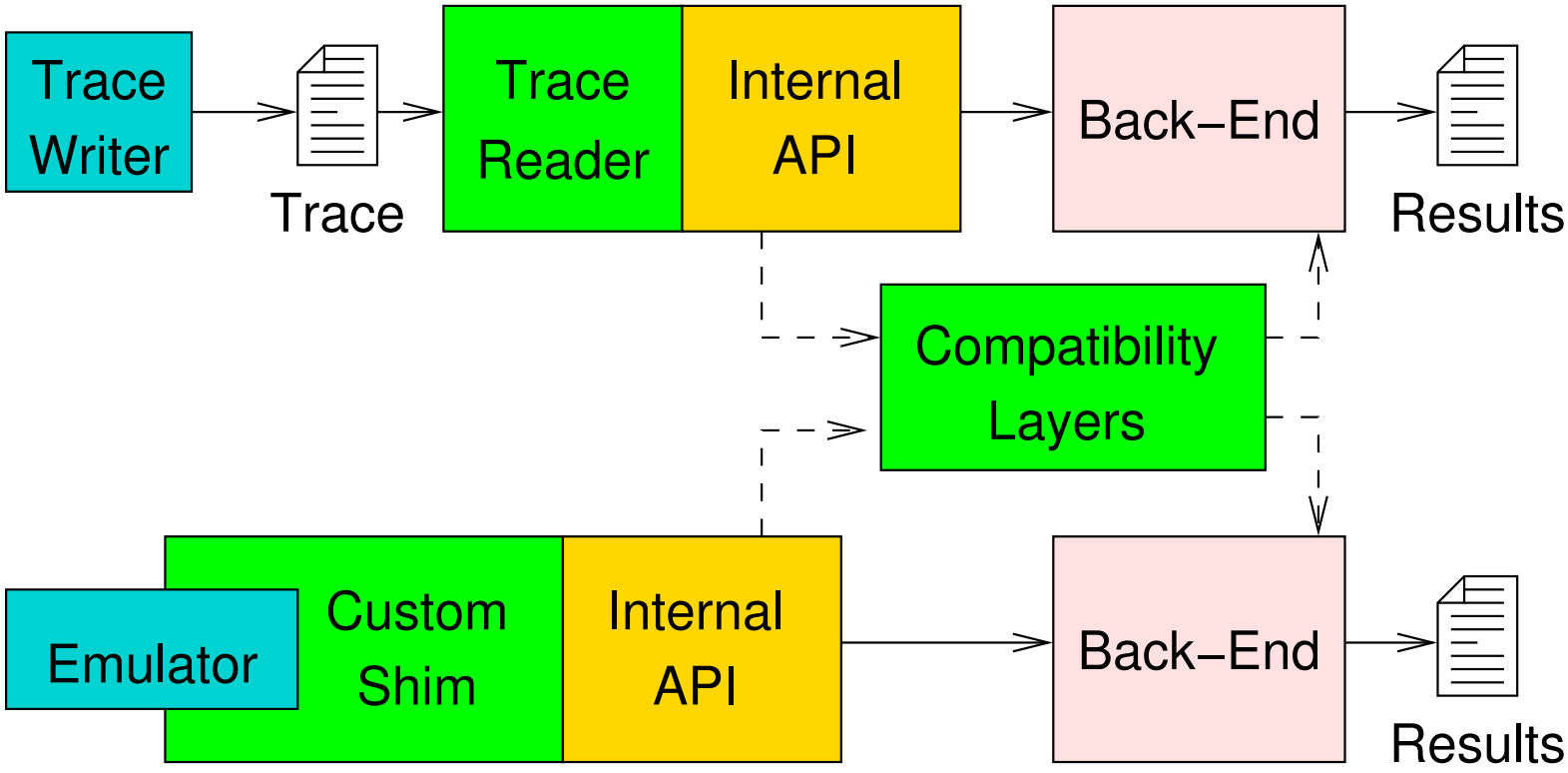
Introduction– Real Front Ends



Realistically:

- Much custom code needs to be written to adapt most off-the-shelf emulators as simulator front ends.

Introduction– Real Front Ends



Realistically:

- Much custom code needs to be written to adapt most off-the-shelf emulators as simulator front ends.
- Each time this is done, yet another simulator-specific front end is created.

Introduction– Front Ends

How do we make new simulator front ends closer to the ideal?

Introduction– Front Ends

How do we make new simulator front ends closer to the ideal?

- Provide an API that does not change unnecessarily beacuse of the type of front end or the instruction set.

Introduction– Front Ends

How do we make new simulator front ends closer to the ideal?

- Provide an API that does not change unnecessarily beacuse of the type of front end or the instruction set.
- Enable the construction of multithreaded simulators.

Introduction– Front Ends

How do we make new simulator front ends closer to the ideal?

- Provide an API that does not change unnecessarily beacuse of the type of front end or the instruction set.
- Enable the construction of multithreaded simulators.
- Provide sufficient control and detail in the API to make it useable with most back ends.

The QSim Simulator Front End

QSim

- We have built a simlator front end based on QEMU[1] that aims to address these issues. It currently:

[1] http://www.qemu.org

QSim

- We have built a simlator front end based on QEMU[1] that aims to address these issues. It currently:
  - Runs unmodified binaries on a lightly-modified Linux guest.

[1] http://www.qemu.org

QSim

- We have built a simlator front end based on QEMU[1] that aims to address these issues. It currently:
  - Runs unmodified binaries on a lightly-modified Linux guest.
  - Enables the construction of multithreaded simulators.

[1] http://www.qemu.org

QSim

- We have built a simlator front end based on QEMU[1] that aims to address these issues. It currently:
  - Runs unmodified binaries on a lightly-modified Linux guest.
  - Enables the construction of multithreaded simulators.
  - Has full support for 32-bit x86 guests. (Port to 64-bit x86 weeks from completion; port to ARM in early coding stages.)

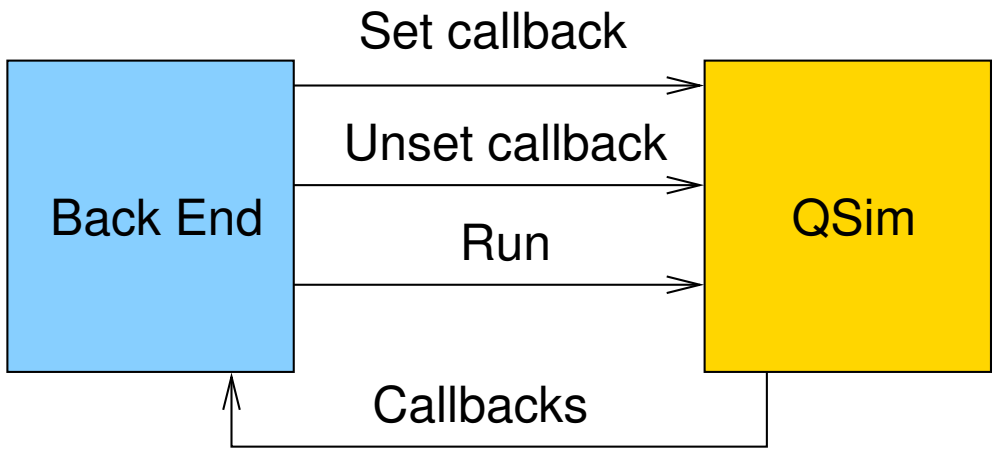
[1] http://www.qemu.org

QSim

- We have built a simlator front end based on QEMU[1] that aims to address these issues. It currently:
  - Runs unmodified binaries on a lightly-modified Linux guest.
  - Enables the construction of multithreaded simulators.
  - Has full support for 32-bit x86 guests. (Port to 64-bit x86 weeks from completion; port to ARM in early coding stages.)
  - Enables parallel simulation.

[1] http://www.qemu.org

QSim– API Overview



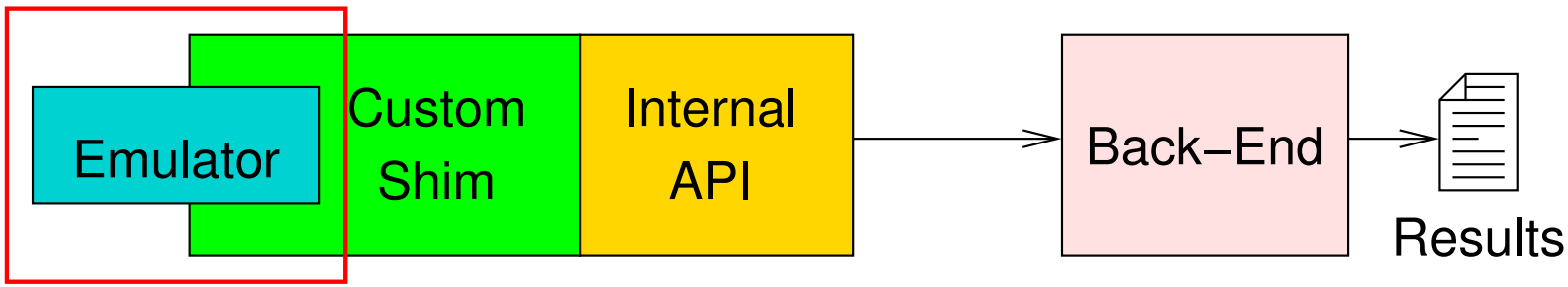
A simplified diagram of the QSim API.

run(i, j)	Advance guest CPU i by j instructions.
set_*_callback(x)	Set callbacks.
unset_*_callback(h)	Unset callbacks by handle.

Callback types include: instruction, register access, memory access, interrupt

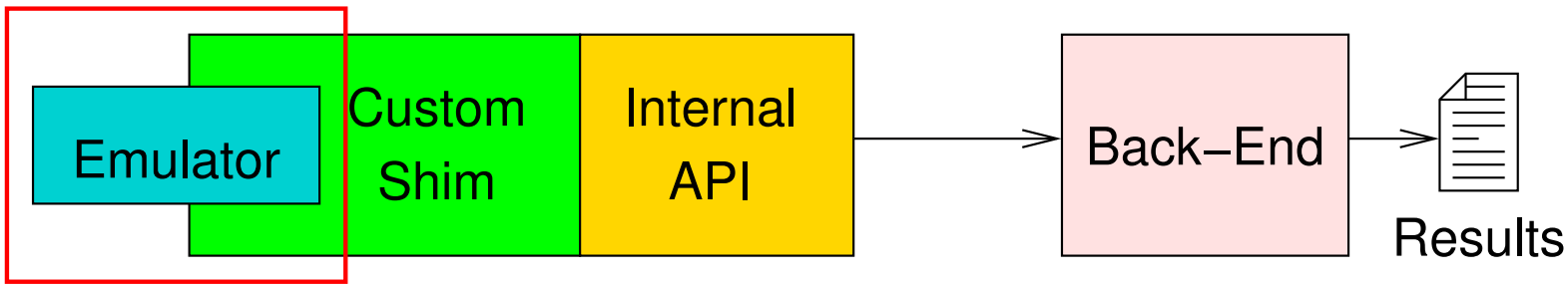
QSim– How is it “Universal”?

Typical emulator used as a simulator front end:



QSim– How is it “Universal”?

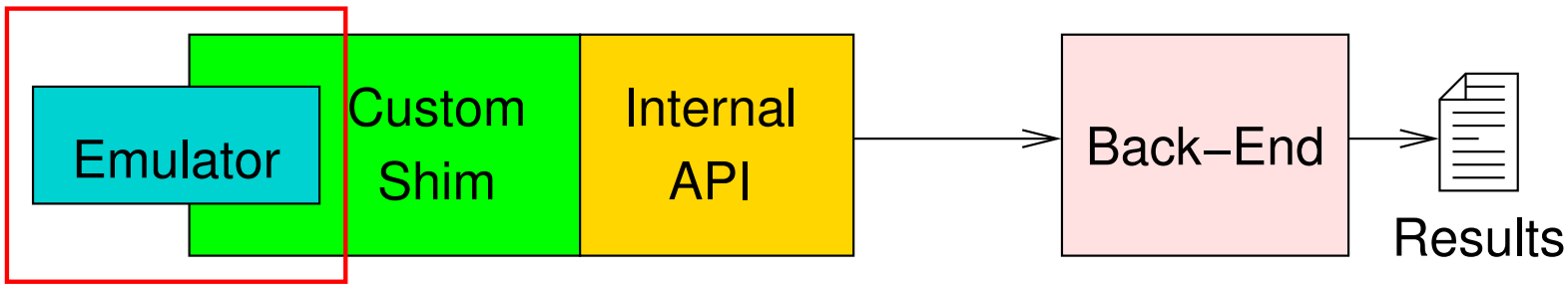
Typical emulator used as a simulator front end:



- Off-the-shelf open-source emulators like QEMU provide most of the code needed to build a front end but are incomplete.

QSim– How is it "Universal"?

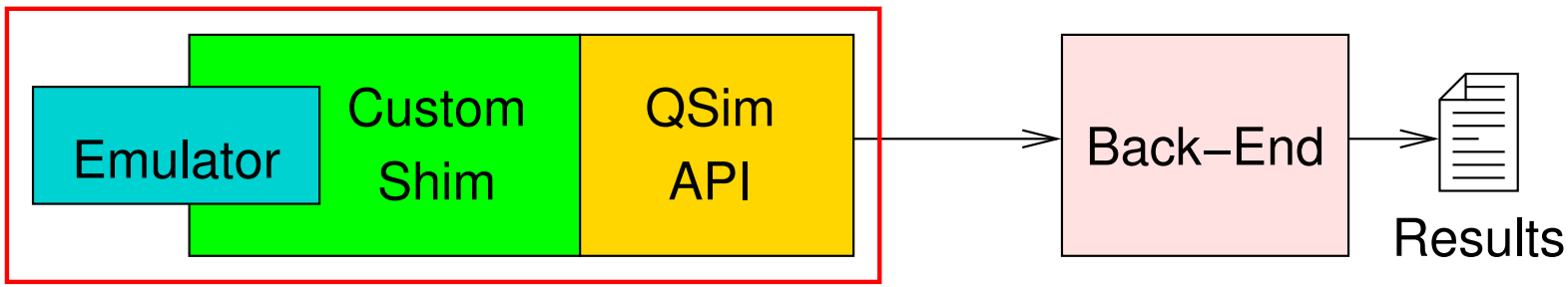
Typical emulator used as a simulator front end:



- Off-the-shelf open-source emulators like QEMU provide most of the code needed to build a front end but are incomplete.
- Simulation projects like PTLSim and FAST have modified QEMU heavily to create their front ends.

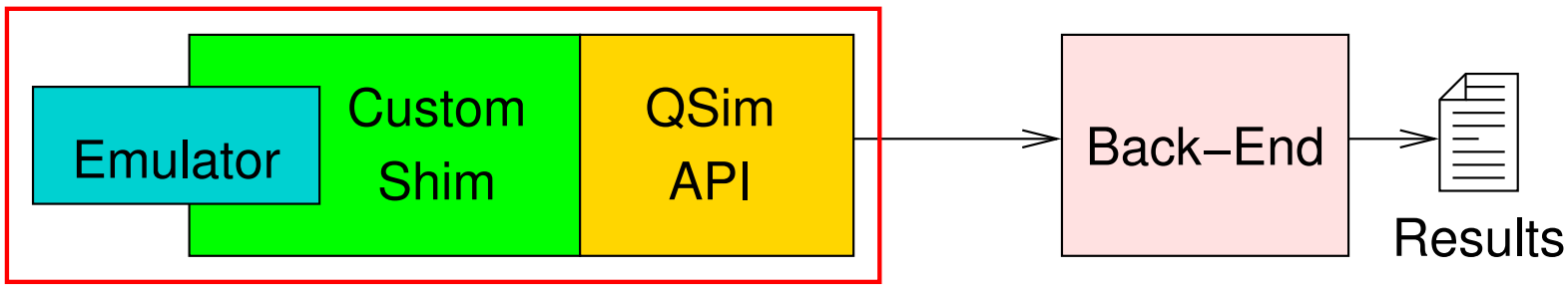
QSim– How is it “Universal”?

QSim:



QSim– How is it "Universal"?

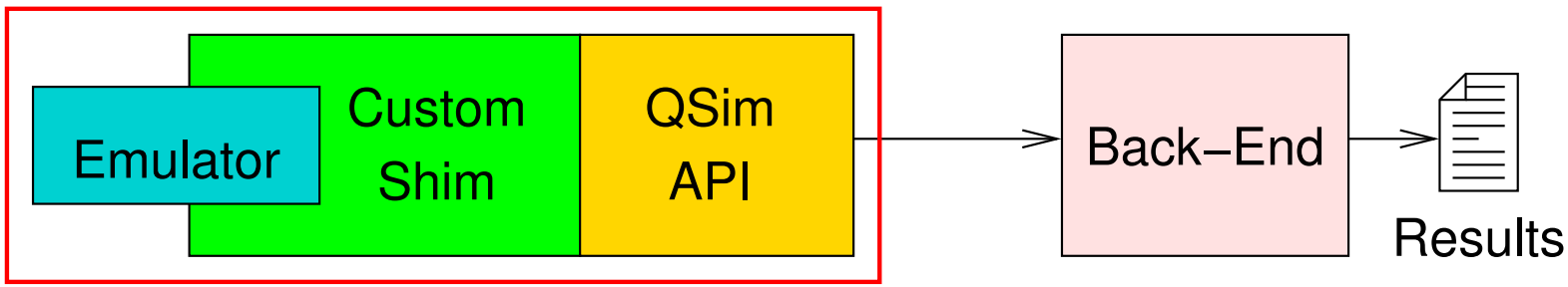
QSim:



- QSim has done this yet again, but with compatibility with a diverse set of back ends in mind.

QSim– How is it "Universal"?

QSim:



- QSim has done this yet again, but with compatibility with a diverse set of back ends in mind.
- Similarly, the API is the same no matter what the instruction set.

QSim– Parallelism

- The **run**() function can be called simultaneously for two different guest CPUs from different host threads.

QSim– Parallelism

- The **run**() function can be called simultaneously for two different guest CPUs from different host threads.
- This enables parallel emulation of multithreaded guests, as well as multithreaded back ends.

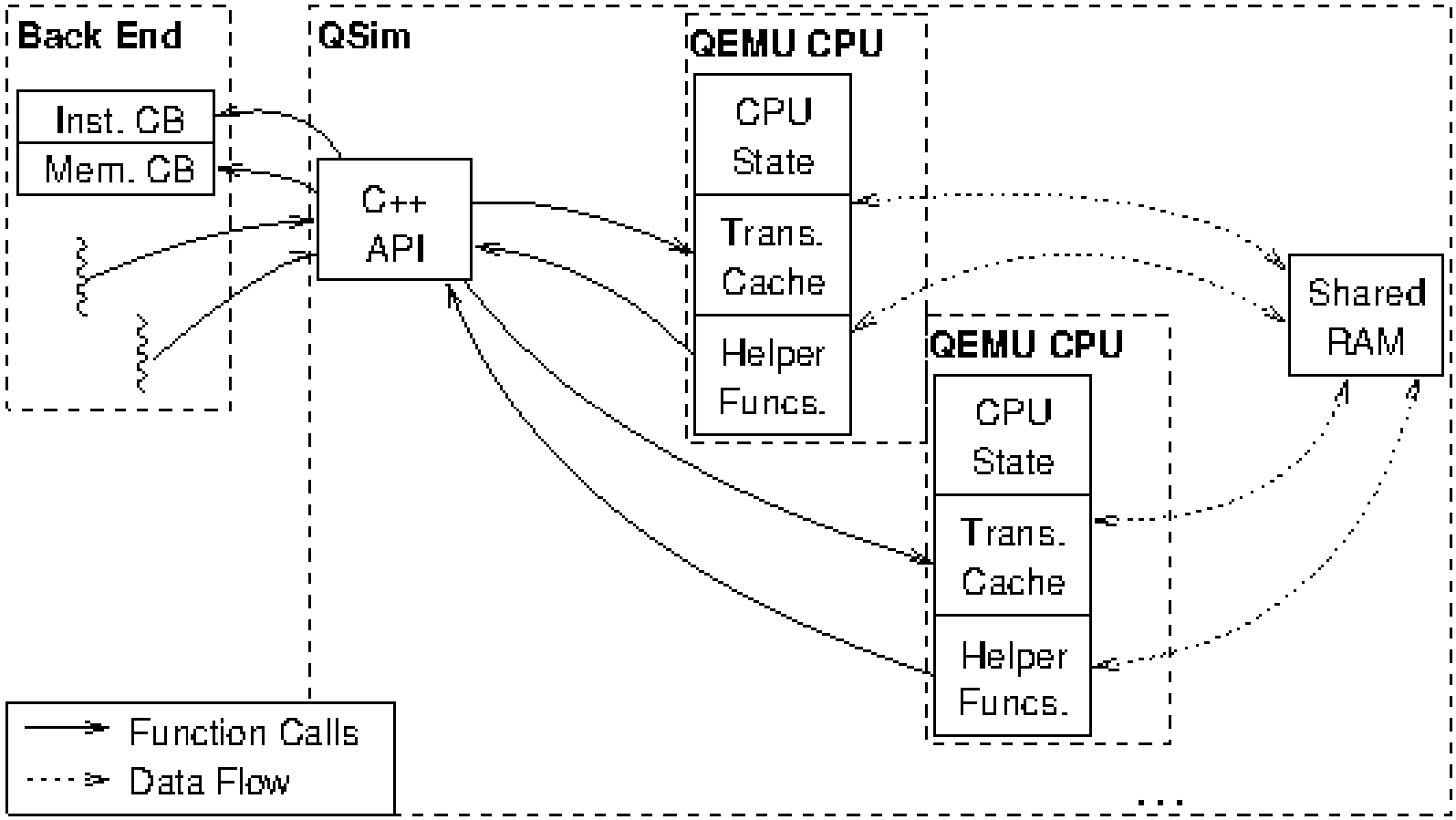
QSim– Parallelism

- The **run**() function can be called simultaneously for two different guest CPUs from different host threads.
- This enables parallel emulation of multithreaded guests, as well as multithreaded back ends.
- Since back ends tend to use more CPU time than front ends, thread safety is more important than parallel emulation (both are provided by QSim).

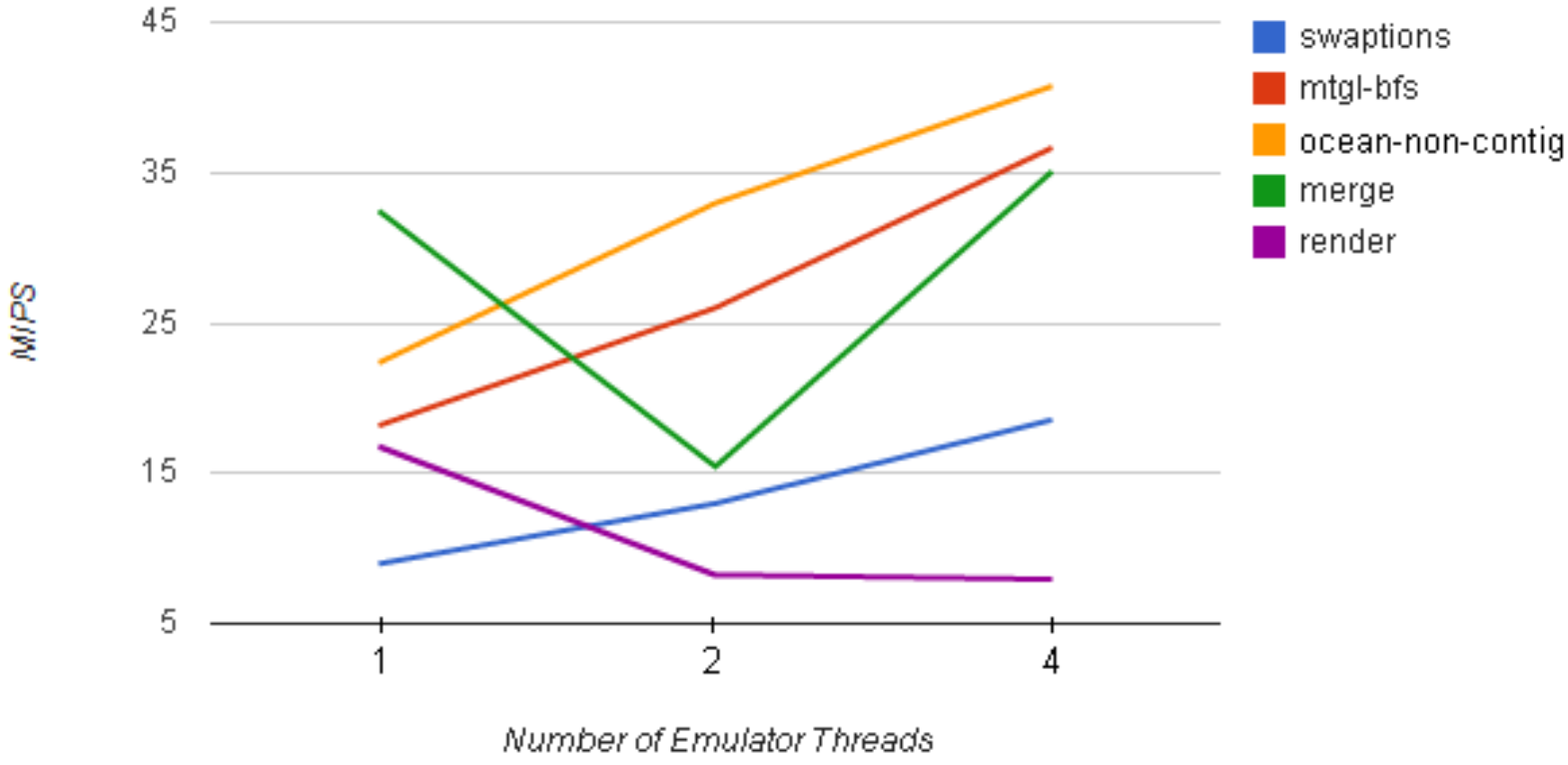
QSim– Parallelism

- The **run**() function can be called simultaneously for two different guest CPUs from different host threads.
- This enables parallel emulation of multithreaded guests, as well as multithreaded back ends.
- Since back ends tend to use more CPU time than front ends, thread safety is more important than parallel emulation (both are provided by QSim).
- Up to 512 guest cores have been demonstrated running on up to 512 host threads.

QSim– Software Architecture



QSim– Parallel Scaling



QSim– Performance

The following represents the performance of QSim with empty callbacks. With typical simulation speeds measured in thousands of instructions per second, QSim will not likely be the bottleneck.

Benchmark	**Slowdown**	**MIPS**
swaptions	259x	18.5
mtgl-bfs	387x	36.6
ocean-non-contig	267x	40.7

QSim– Relation to QEMU

How are QEMU and QSim related?

QEMU	**QSim**

QSim– Relation to QEMU

How are QEMU and QSim related?

QEMU	**QSim**
Emulator	Simulator Front-End

QSim– Relation to QEMU

How are QEMU and QSim related?

QEMU	**QSim**
Emulator	Simulator Front-End
Standalone	Built on QEMU

QSim– Relation to QEMU

How are QEMU and QSim related?

QEMU	**QSim**
Emulator	Simulator Front-End
Standalone	Built on QEMU
Full-system	CPU and RAM only

QSim– Relation to QEMU

How are QEMU and QSim related?

QEMU	**QSim**
Emulator	Simulator Front-End
Standalone	Built on QEMU
Full-system	CPU and RAM only
CPUs serialized	CPUs in parallel

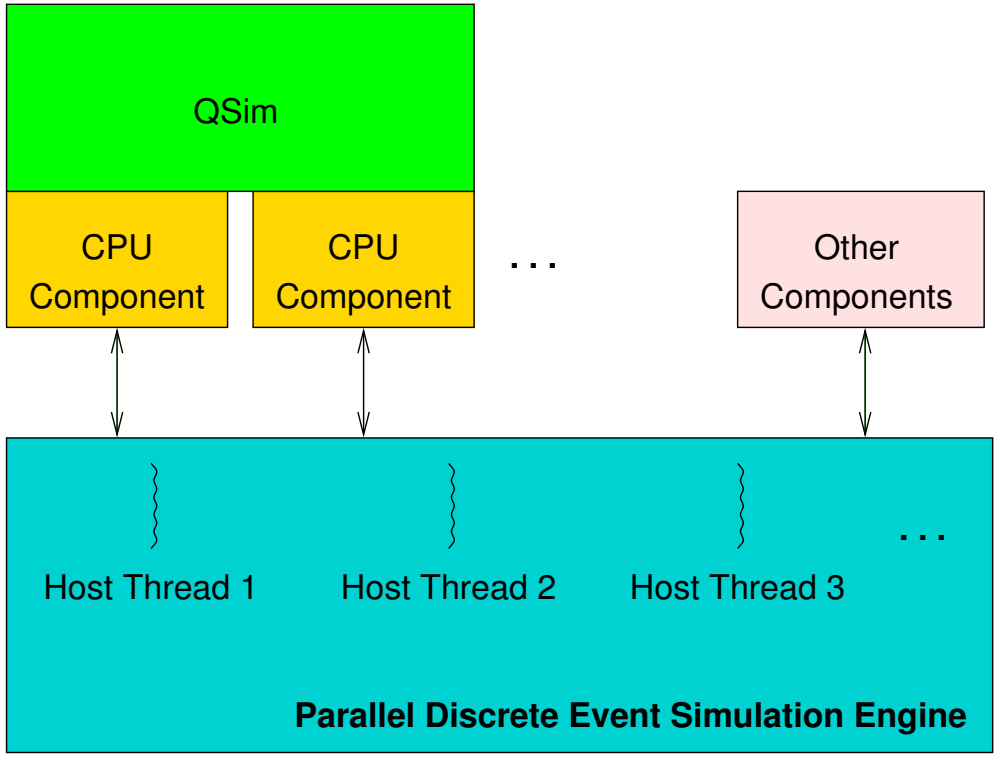
QSim– Relation to QEMU

How are QEMU and QSim related?

QEMU	**QSim**
Emulator	Simulator Front-End
Standalone	Built on QEMU
Full-system	CPU and RAM only
CPUs serialized	CPUs in parallel
Program	Library

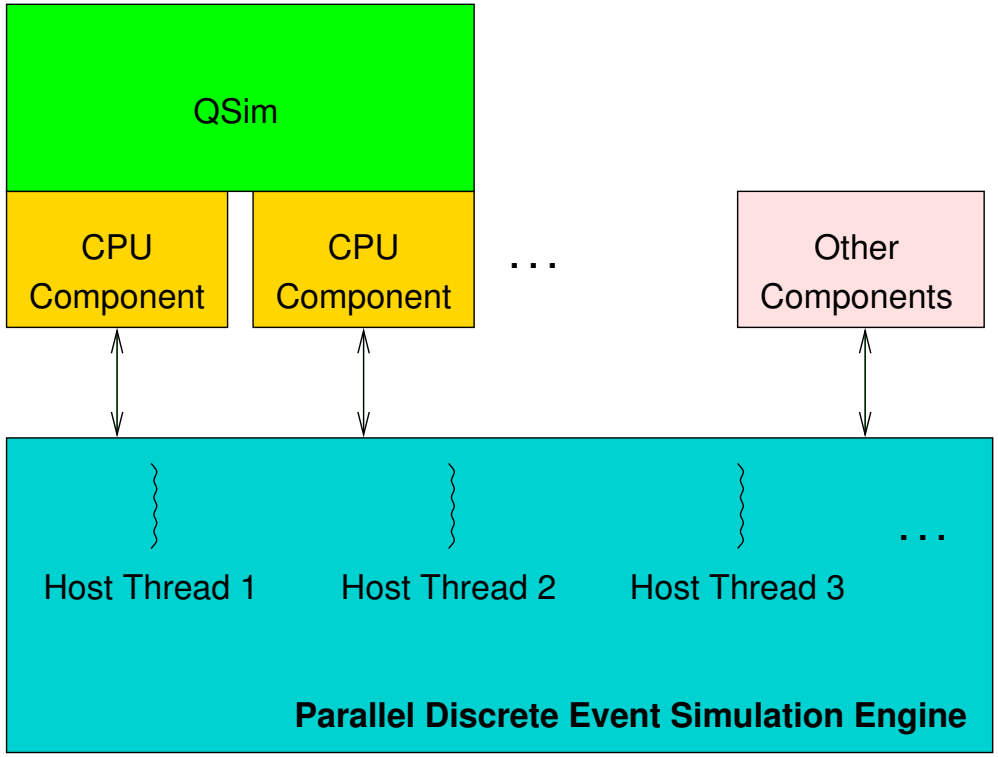
Back Ends

Back Ends– The Canonical Example



This is the kind of simulation QSim was designed for.

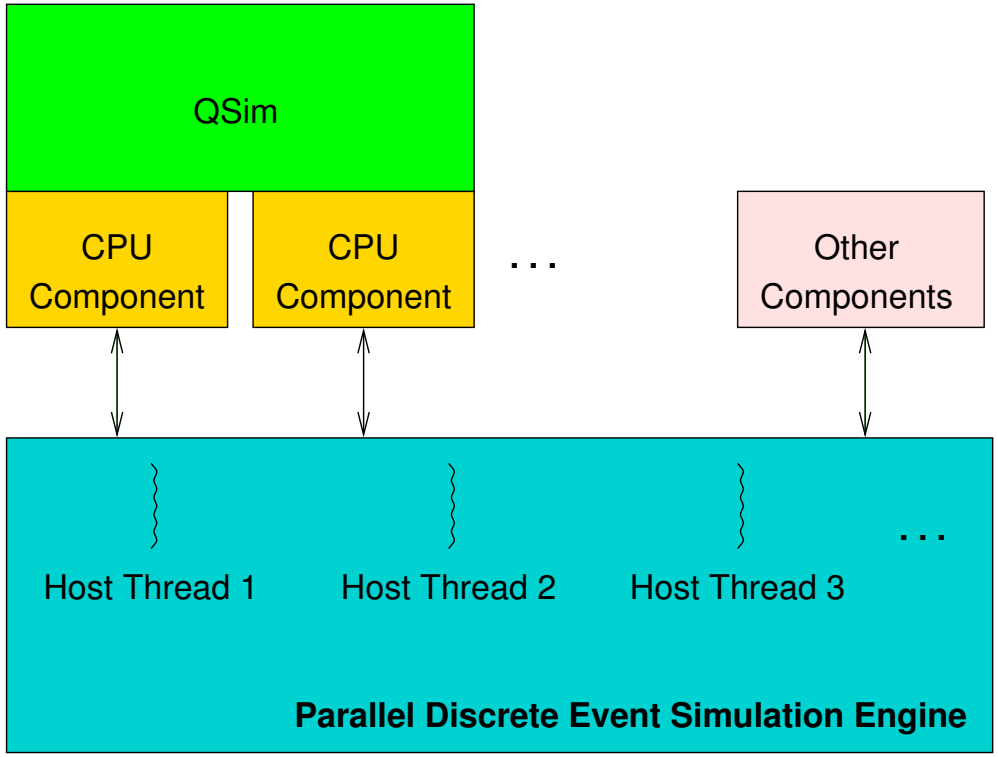
Back Ends– The Canonical Example



This is the kind of simulation QSim was designed for.

- QSim feeds instructions into CPU timing models that are part of a larger simulation infrastructure.

Back Ends– The Canonical Example



This is the kind of simulation QSim was designed for.

- QSim feeds instructions into CPU timing models that are part of a larger simulation infrastructure.
- A parallel discrete event simulation engine keeps track of events and ensures correctness.

Back Ends– Others

Other back ends that have been built for QSim include:

Back Ends– Others

Other back ends that have been built for QSim include:

- A binary trace writer, which was built along with a trace reader library that exports the QSim API.

Back Ends– Others

Other back ends that have been built for QSim include:

- A binary trace writer, which was built along with a trace reader library that exports the QSim API.
- A serial universal processor emulator, simplesim.

Back Ends– Others

Other back ends that have been built for QSim include:

- A binary trace writer, which was built along with a trace reader library that exports the QSim API.
- A serial universal processor emulator, simplesim.
  - A demonstration vehicle; breaks instructions into plausible micro-ops regardless of instruction set.

Back Ends– Others

Other back ends that have been built for QSim include:

- A binary trace writer, which was built along with a trace reader library that exports the QSim API.
- A serial universal processor emulator, simplesim.
  - A demonstration vehicle; breaks instructions into plausible micro-ops regardless of instruction set.
- An interactive OS/application debugger.

Back Ends– Others

Other back ends that have been built for QSim include:

- A binary trace writer, which was built along with a trace reader library that exports the QSim API.
- A serial universal processor emulator, simplesim.
  - A demonstration vehicle; breaks instructions into plausible micro-ops regardless of instruction set.
- An interactive OS/application debugger.
- Visualization utilities.

Summary

This has been:

[2] http://www.cdkersey.com/qsim-web/

Summary

This has been:

- An appeal for consistent front end/back end APIs.

[2] http://www.cdkersey.com/qsim-web/

Summary

This has been:

- An appeal for consistent front end/back end APIs.
- A plea for parallel front ends.

[2] http://www.cdkersey.com/qsim-web/

Summary

This has been:

- An appeal for consistent front end/back end APIs.
- A plea for parallel front ends.
- A look at how we might narrow the simulation gap.

[2] http://www.cdkersey.com/qsim-web/

Summary

This has been:

- An appeal for consistent front end/back end APIs.
- A plea for parallel front ends.
- A look at how we might narrow the simulation gap.
- An invitation to try QSim[2].

[2] http://www.cdkersey.com/qsim-web/

Acknowledgements

The authors are grateful to Paolo Faraboschi and Daniel Ortega for their suggestions and guidance in getting QSim started. This work was supported by the National Science Foundation under grant CNS855110, Sandia National Laboratories, and HP Laboratories.