B4 Performance Optimization and Simple Cache

> **! Update ysyxSoC**
>
> The SoC team has provided a 32-bit tape-out SoC, and we have also modified `ysyxSoC` to 32-bit on 2024/07/26 at 13:00:00, to help everyone test locally before connecting to the tape-out SoC. If you obtained the `ysyxSoC` code before the above time, please perform the following steps:
>
> 1. Execute the following command to fetch the new 32-bit `ysyxSoC` :
>
>    ```sh
>    cd ysyxSoC
>    git pull origin master # You may need to resolve some code conflicts
>    ```
>
> 2. Change the AXI data width of the top NPC to 32-bit and remove the related data width conversion code.
> 3. Re-run simulations using the 32-bit `ysyxSoC` environment.

After connecting to ysyxSoC, your designed NPC can now correctly interact with various devices, meaning it is ready to participate in the tape-out from a functional perspective. Following the principle of "complete first, perfect later" in system design, we can now discuss performance optimization.

Although we call it performance optimization, this is just the final goal. In a complex system, we will face many choices, such as: Which areas are worth optimizing? What optimization methods should we use? What are the expected benefits? What are the costs of these methods? If we put in a lot of effort and only achieve a 0.01% performance improvement, this is certainly not what we want. Therefore, rather than blindly writing code, we need a scientific approach that guides us to answer the above questions:

1. Evaluate current performance

2. Identify performance bottlenecks
3. Apply appropriate optimization methods
4. Evaluate the optimized performance and compare the performance gains with expectations

Performance Evaluation

Before we can talk about optimization, we first need to know how the current system is performing. Thus, we need a quantitative performance metric, rather than judging "how well it runs" based on our feelings. Using this metric to evaluate the current system is the first step in performance optimization.

Benchmark Program Selection

So which programs should we evaluate? There are various programs, and it is unrealistic to evaluate all of them, so we need to select some representative programs. "Representative" means that the performance benefits of optimization techniques on these programs should align with those in real-world application scenarios.

Here, we mention "application scenarios", which means the trends for performance benefits may vary across different scenarios. This implies that different application scenarios require different representative programs, leading to various benchmarks. For example, Linpack represents the supercomputing scenario, MLPerf represents machine learning training, CloudSuite represents cloud computing, and Embench represents embedded systems. For general computing scenarios, the most famous benchmark is SPEC CPU, which evaluates CPU general-purpose computational power. SPEC (Standard Performance Evaluation Corporation) is an organization that defines and maintains benchmarks for evaluating computer systems, and has published various benchmarks for different scenarios. In addition to SPEC CPU, there are also benchmarks for graphics, workstations, high-performance computing, storage, power consumption, virtualization, etc.

A benchmark typically consists of several sub-items. For example, SPEC CPU 2006's integer test includes the following sub-items:

Sub-item	Description
400.perlbench	Perl spam detection
401.bzip2	bzip compression algorithm

Sub-item	Description
403.gcc	gcc compiler
429.mcf	Large public transportation optimization problem
445.gobmk	Go, AI search problem
456.hmmer	Gene sequence search using Hidden Markov Models
458.sjeng	Chess, AI search problem
462.libquantum	Quantum computation for prime factorization
464.h264ref	H.264 video encoding of YUV format source files
471.omnetpp	Large CSMA/CD protocol Ethernet simulation
473.astar	A* pathfinding algorithm
483.xalancbmk	XML to HTML format conversion

In addition to integer tests, SPEC CPU 2006 also includes floating-point tests, covering areas such as fluid dynamics, quantum chemistry, biomolecules, finite element analysis, linear programming, ray tracing, computational electromagnetics, weather forecasting, and speech recognition.

Of course, benchmarks need to evolve with the times to represent the programs of the new era. As of 2024, SPEC CPU has evolved through six versions, starting from 1989, 1992, 1995, 2000, 2006, and finally the latest version in 2017. SPEC CPU 2017 introduced new programs representing new application scenarios, such as biomedical imaging, 3D rendering and animation, and AI Go programs using Monte Carlo Tree Search (likely influenced by AlphaGo in 2016).

> **CoreMark and Dhrystone are not good benchmarks**
>
> CoreMark and Dhrystone are synthetic programs, meaning they consist of several code snippets pieced together. For example, CoreMark consists of linked list operations, matrix multiplication, and state machine transitions; Dhrystone is made up of string operations.
>
> The major problem with synthetic programs as benchmarks is their weak representativeness: What application scenarios do CoreMark and Dhrystone represent? Compared to the various real-world applications in SPEC CPU 2006, CoreMark's code snippets are barely C-language homework; Dhrystone is even further

> from real applications, with very simple code (using short string literals), and under modern compilers, the code in the loop body is likely to be deeply optimized (remember `pattern_decode()` in NEMU), causing the evaluation results to be inflated, failing to objectively reflect system performance. **This article** provides a detailed analysis of Dhrystone's flaws as a benchmark.
>
> Ironically, many CPU vendors still use CoreMark or Dhrystone results to indicate their product performance, even for products supposedly designed for high-performance scenarios. **Turing Award winner David Patterson, a pioneer of computer architecture, commented on Embench**, stating that Dhrystone is long outdated and should be retired. In fact, the first version of Dhrystone was released in 1984, and it has not been maintained or updated since 1988. Compared to the 1980s, the computer field has drastically changed, with updated applications, mature compiler technologies, and much more powerful hardware. Using a 40-year-old benchmark to evaluate today's computers is certainly questionable.

For the "One Student One Chip" teaching scenario, SPEC CPU's programs are a bit too realistic: On one hand, they are large-scale, requiring hours to run even on an x86 real machine; On the other hand, they need to run in a Linux environment, meaning we first need to design a CPU that can correctly boot Linux, and only then can we run the SPEC CPU benchmark.

In contrast, we want a benchmark suitable for teaching scenarios that meets the following conditions:

- Not too large in scale, with execution times under 2 hours in simulators or RTL simulation environments
- Can run in a bare-metal environment, without needing Linux
- The program is representative, unlike CoreMark and Dhrystone which are synthetic programs

In fact, the microbench integrated in `am-kernels` is a good choice. On one hand, microbench provides test sets of various sizes, where simulators can use the `ref` size and RTL simulation environments can use the `train` size; On the other hand, microbench is an AM program, so it can run without Linux; Additionally, microbench includes 10 sub-items, covering sorting, bit operations, language interpreters, matrix calculations, prime number generation, A* algorithm, maximum network flow, data compression, MD5 checksum, etc. Therefore, when future lectures refer to performance evaluation but do not specify the corresponding benchmark, it will refer to the `train` size of microbench by default.

If the processor's application scenario is clear, such as running Super Mario, then we can directly use Super Mario as the benchmark, equating "Super Mario's gameplay experience" with the standard of "good performance". Unlike microbench, Super Mario is a program that never finishes running, so we can use FPS as a quantitative metric to evaluate, rather than execution time.

Finding Performance Bottlenecks

Performance Formula and Optimization Directions

We can measure the running time of the benchmark to obtain the system's performance. But the running time is a single data point, and it is difficult to identify performance bottlenecks directly from it, so we need more detailed data.

In fact, we can break down the program's running time into the following three factors:

```
        time      inst     cycle     time
perf = ------- = ------ * ------- * -------
        prog      prog      inst     cycle
```

The goal of performance optimization is to reduce the program's running time, which means minimizing each of these factors. This reveals three optimization directions.

The first optimization direction is to reduce the `number of instructions executed by the program (i.e., dynamic instruction count)`. Possible measures include:

1. Modifying the program to use more optimal algorithms.
2. Using better compiler optimization strategies. For example, in gcc, in addition to using common optimization flags like `-O3`, `-Ofast`, we can also fine-tune the compilation options for the target program. There are about 600 compiler options in gcc related to code quality, and choosing the right ones can significantly reduce dynamic instruction count. For example, when working on a project, yzh compiled CoreMark with just `-O3`, and the dynamic instruction count for 10 runs was about 3.12 million; by enabling some targeted compiler options, the dynamic instruction count dropped to about 2.25 million, significantly improving performance.
3. Designing and using more complex instruction sets. We know that CISC instruction sets contain complex instructions, and if the compiler uses these, it can reduce dynamic instruction count. Additionally, custom dedicated instructions can be added to the processor and used by the program.

The second optimization direction is to reduce the `cycles per instruction on average (CPI)`. Another way of saying this is to increase IPC (Instructions Per Cycle). This metric reflects the processor's microarchitecture design: a powerful processor executes more instructions per cycle. Thus, microarchitecture optimization typically aims to improve IPC, speeding up program execution. Microarchitecture optimization has different directions, which we will briefly discuss later.

The third optimization direction is to reduce the `time per cycle`, or increase the cycles per unit time, which is the circuit's frequency. Possible optimization measures include:

1. Optimizing the front-end design of digital circuits to reduce critical path logic delays.
2. Optimizing the back-end design of digital circuits to reduce critical path wiring delays.

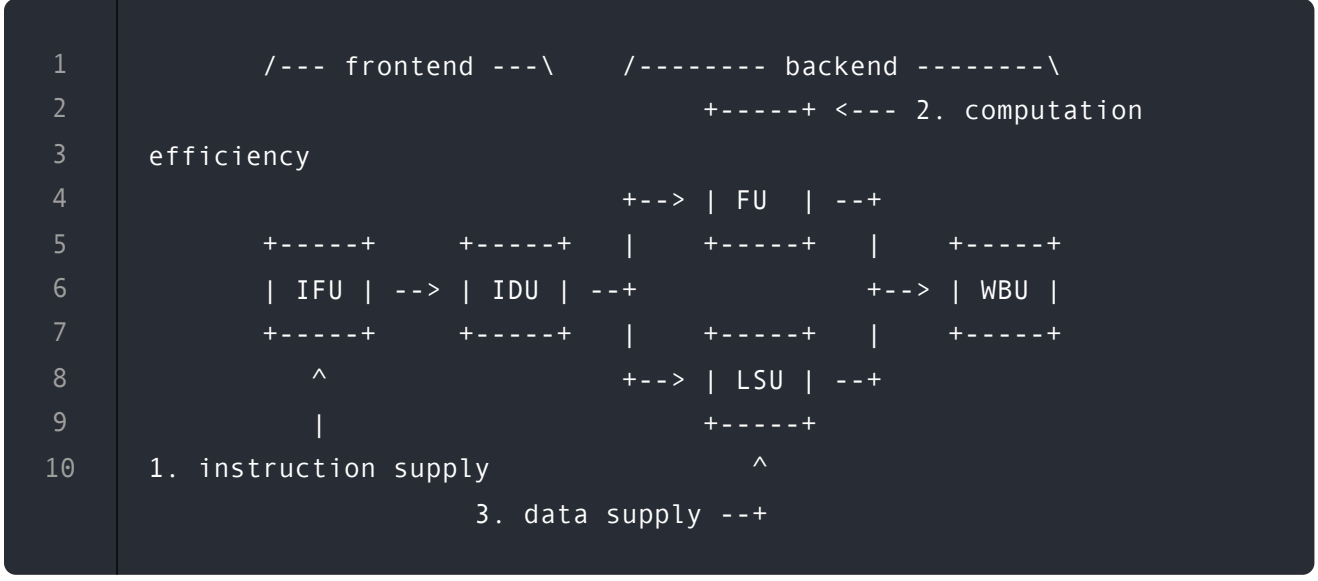
If we can quantify these three factors, we can better assess the potential of these optimization directions, which will help guide us in finding the performance bottleneck. Fortunately, these metrics are not difficult to obtain:

- Dynamic instruction count can be directly counted in the simulation environment.
- With the dynamic instruction count, we can calculate IPC by counting the cycle count.
- Circuit frequency can be found in the synthesizer's report.

> **Count IPC**
>
> Try to count IPC in the simulation environment.
>
> Actually, when we implemented the bus, we asked you to evaluate the program's running time using the performance formula above. However, we calculated it using `program cycles / frequency` at that time, and the performance formula above simply breaks down `program cycles` into two factors. Nevertheless, this breakdown can still provide us with more detailed information, since `program cycles` depend on both the program and processor, while `cycles per instruction` depends only on the processor's capabilities.

Simple Processor Performance Model

Even though IPC is easy to count, just as running time can't guide us on how to optimize it, the measured IPC also can't guide us on how to optimize it. To find performance bottlenecks, we need to analyze the factors that influence IPC, just as we did for running time. For this, we need to re-examine how the processor executes instructions.



The above is a simple processor block diagram, and we have previously understood it from a functional perspective. Now, we need to look at it from a performance perspective. We can divide the processor into the front-end and back-end, where the front-end handles instruction fetch and decode, and the remaining modules are in the back-end, responsible for executing instructions and updating the processor state. Note that the front-end and back-end division in processors is different from the digital circuit front-end and back-end design mentioned earlier. In fact, both the processor's front-end and back-end design belong to the digital circuit's front-end design phase.

To improve processor execution efficiency, we need to ensure:

1. The front-end guarantees instruction supply. If the front-end can't fetch enough instructions, the processor's computational power can't be fully utilized. Because each instruction execution requires an instruction fetch, the front-end's instruction supply capacity affects the execution efficiency of all instructions.
2. The back-end guarantees computation efficiency and data supply.
   - For most computational instructions, their execution efficiency depends on the efficiency of the corresponding functional unit. For example, the efficiency of multiplication and division instructions depends on the efficiency of the multiplier/divider. Similar effects apply to floating-point operations and floating-point units (FPU).
   - For memory access instructions, their execution efficiency depends on the efficiency of the LSU. Specifically, for load instructions, the processor must wait for data from memory before writing it back to the register file. This means the execution efficiency of load instructions depends on LSU and memory's data supply capabilities. Store instructions are special because they don't require writing to the register file; the processor doesn't need to wait for the data to be completely written to memory. In high-performance processors, a store buffer is often designed, where the processor

considers a store instruction complete after writing to the store buffer, and the store buffer handles the actual writing to memory later. However, this increases the complexity of the processor design, such as ensuring that load instructions check whether the latest data is in the store buffer.

So how do we quantify instruction supply, computation efficiency, and data supply? In other words, we want to understand if modules like the IFU and LSU are running at full speed when the processor runs a given benchmark. For this, we need to gather more information.

Performance Events and Performance Counters

To quantitatively assess the processor's instruction supply, computational efficiency, and data supply, we need to further understand the detailed factors that influence them. Take instruction supply as an example. How do we determine the strength of instruction supply capability? The direct indicator of instruction supply capability is whether the IFU has fetched the instruction. To this end, we can treat “IFU fetching instructions” as an event and count the frequency of this event occurring.
If this event occurs frequently, the instruction supply capability is strong; otherwise, it is weak.

These events are called performance events (performance event), and through them,
we can convert some of the more abstract performance metrics in the performance model into specific events on the circuit. Similarly, we can also count the frequency of the event “LSU fetching data” to measure the strength of data supply capability;
and count the frequency of the event “EXU completing computation” to measure the efficiency of computation.

To count the frequency of performance events, we only need to add some counters to the hardware,
and increment the counter value by 1 when a performance event occurs. These counters are called performance counters.
With performance counters, we can observe “where the program spends its time running on the processor,”
which is equivalent to profiling the processor's internal operations.

Detecting the occurrence of performance events on the circuit is not difficult; we can use the handshake signals of the bus mechanism for detection. For example, when the R channel handshakes during IFU instruction fetch, it indicates that the IFU has received data returned from the AXI bus, thereby completing an instruction fetch operation.

Therefore, when the R channel handshakes, we can increment the corresponding performance counter by 1.

Add performance counters

Attempt to add some performance counters in the NPC, including at least the following performance counters for performance events:

- IFU fetches an instruction
- LSU fetches data
- EXU completes the computation
- Decodes various types of instructions, such as computational instructions, memory access instructions, CSR instructions, etc.

Performance counters are essentially implemented by circuitry. As the number of performance counters increases,
they will occupy an increasingly larger circuit area, potentially affecting critical paths in the circuit. Therefore, we do not require performance counters to be included in the tape-out. You can use them solely in the simulation environment: You can implement performance counters using RTL and output their values at the end of simulation via methods like `$display()`, then configure the synthesis process to avoid instantiating them; Alternatively, you can connect the performance event detection signals to the simulation environment via DPI-C and implement performance counters in the simulation environment.
This way, you can freely add performance counters without worrying about affecting the circuit area and frequency.

After implementation, try running the microbench test scale and collect the performance counter results. If your implementation is correct, the statistical results of different performance counters with similar semantics should be consistent. For example, the total number of different categories of instructions decoded should be consistent with the number of instructions fetched by the IFU and also with the dynamic instruction count. Try to identify more consistent relationships and verify whether these relationships hold true.

Sometimes, we are more concerned with when an event does not occur and why it does not occur, rather than when it does occur.
For example, we are actually more concerned with when the IFU cannot fetch instructions and why it cannot fetch instructions.

Understanding the underlying reasons helps us identify bottlenecks in instruction supply, which in turn provides guidance for improving the processor's instruction supply.
We can define “event not occurring” as a new event and add performance counters for the new event.

Add performance counters (2)

Add more performance counters in the NPC and attempt to analyze the following issues:

- What percentage of instructions does each category account for? How many cycles does each category average to execute?
- What are the reasons why the IFU cannot fetch instructions? What are the probabilities of these reasons causing the IFU to fail to fetch instructions?
- What is the average memory access latency of the LSU?

Performance counter trace

The usage methods of performance counters described earlier all involve outputting and analyzing the results after simulation. If we output the values of performance counters every cycle, we can obtain a performance counter trace! Based on this trace, using some plotting tools (such as Python's matplotlib plotting library), we can plot the performance counter values over time, visualize the changes in performance counters during the simulation, and thus better judge whether the changes in performance counters are as expected.

Amdahl's Law

Performance counters can provide quantitative guidance for optimizing processor microarchitecture.
So, where exactly is the performance bottleneck? Which optimization efforts are worthwhile? What is the expected performance gain from optimization efforts?
We need to answer these questions before undertaking specific optimizations, to help us avoid optimization efforts with low expected performance gains, and instead focus more time on optimization efforts with high expected gains. This may sound like predicting the future, but Amdahl's Law can provide the answer.

Amdahl's Law was proposed by computer scientist Gene Amdahl in 1967, and states:

```
The overall performance improvement gained by optimizing a single
part
of a system is limited by the fraction of time that the improved
part
is actually used.
```

Assuming that the proportion of time that a certain part of the system is actually used is `p`, and the acceleration ratio of that part after optimization is `s`, then the acceleration ratio of the entire system is `f(s) = 1 / (1 - p + p / s)`, which is the formula for Amdahl's Law.

For example, the execution process of a program is divided into two independent parts, A and B, where A accounts for 80% and B accounts for 20%.

- If B is optimized by a factor of 5, the overall program's speedup ratio is `1 / (0.8 + 0.2 / 5) = 1.1905`;
- If B is optimized by a factor of 5000, the overall program's speedup ratio is `1 / (0.8 + 0.2 / 5000) = 1.2499`;
- If A is optimized by a factor of 2, the overall program acceleration ratio is `1 / (0.2 + 0.8 / 2) = 1.6667`.

```
<------- A --------><-B->
++++++++++++++++++++ooooo   Original program

++++++++++++++++++++o       Optimize B by a factor of 5

++++++++++++++++++++        Optimize B by 5000 times, resulting in a
very short runtime for the optimized B

++++++++++ooooo             Optimize A by 2 times
```

Generally speaking, optimizing by 5000 times requires significantly more effort than optimizing by 2 times, but Amdahl's law tells us that optimizing B by 5000 times is less effective than optimizing A by 2 times. This counterintuitive conclusion tells us that we cannot simply consider the acceleration ratio of a particular component, but must also consider its time proportion, and evaluate the optimization effect of a technology from the perspective of the entire system. Therefore, for processor performance optimization, it is very important to measure the proportion of a particular optimization target in the runtime using performance counters in advance.

Identify suitable performance bottlenecks using performance counters

Based on the statistics from performance counters, attempt to identify potential optimization targets, then use Amdahl's law to estimate their theoretical performance gains, thereby determining where the system's performance bottlenecks lie.

Professional ethics in computer architecture

There is a widely circulated piece of advice in the software field:

```
Discussing optimization without considering the workload is irresponsible.
```

This means that the selection of optimization schemes must be based on the actual runtime conditions of the workload.

This is especially true in the field of computer architecture, where we cannot rely on intuition to optimize processor designs, making changes wherever we perceive optimization opportunities, as this can easily lead to ineffective solutions and may even result in performance degradation in real-world scenarios. Instead, adopting a suitable design scheme based on evaluation data is the scientific approach.

In fact, Amdahl's Law is easy to understand; without considering professional background, we could even package it as a math application problem for elementary school students to solve. However, we have also seen many beginners “play fast and loose,” which boils down to a lack of relevant professional expertise.

When we learn “One Student One Chip,” it's not just about learning RTL coding,
but more importantly, it's about learning scientific problem-solving methods and developing professional expertise in this field,
so that when we encounter real-world problems in the future, we know how to solve them using the correct approach.

Top-down debugging method

You've debugged many functional bugs, but there's also a type of bug called a performance bug, which doesn't manifest as program errors or crashes, but rather as

program performance that falls short of expectations. Of course, the process of debugging performance bugs shares similarities with the process of performance optimization, both requiring the identification of performance bottlenecks within the system.

In fact, debugging functional bugs and debugging performance bugs also share similarities. When debugging functional bugs, we first see error or crash messages from the program, but simply reading such messages makes it difficult to locate the bug; therefore, we use various levels of trace tools to understand the program's behavior and identify the specific manifestations of the error; then we use tools like GDB or waveforms to conduct detailed analysis at the variable/signal level.

When debugging performance bugs, we first observe the program's runtime, but this does not directly reveal where the performance bottlenecks are; we then decompose the program's runtime into three factors using performance formulas, and examine optimization potential from three directions: compilation, microarchitecture, and frequency; for microarchitecture, relying solely on statistical IPC metrics still makes it difficult to identify performance bottlenecks, Therefore, we need to analyze the factors affecting IPC, dividing the processor into three major components, understanding the processor's instruction execution process through instruction supply, data supply, and computational efficiency, but we require more specific quantitative data; thus, we need to add performance counters to track the occurrence of performance events in each module, and finally use Amdahl's Law to identify the true performance bottleneck.

Debugging both types of bugs involves a similar top-down analysis approach, which is no coincidence, but rather a reflection of abstract thinking in the field of computer systems: abstraction is the only way to understand complex systems. In fact, if you try to debug using GDB/waveforms from the start, you will find it very challenging, as the sheer volume of low-level details makes it difficult to gain a macro-level understanding. Therefore, we need to start from high-level semantics and trace downwards along appropriate paths, narrowing the scope to a very small area at the lower level, which enables us to quickly pinpoint the issue.

Calibrating Memory Access Latency

After connecting the NPC to the ysyxSOC, modules such as the SDRAM controller provide a more realistic memory access process. Imagine if we had counted performance counters

before connecting to the ysyxSOC; due to differences in memory access latency, the results would be significantly different from those obtained after connecting to the ysyxSOC. Different statistical results would guide us toward different optimization directions, but if our goal is to achieve the desired performance, these different optimization directions may not yield the expected results. Therefore, the closer the behavior of the simulation environment aligns with that of the actual chip, the smaller the error in the evaluation results, and the performance improvements achieved through optimizations guided by performance counters will be more accurate.

In fact, the previous ysyxSOC environment assumed that the processor and various peripherals operate at the same frequency: One cycle in Verilator simulation corresponds to one cycle in both the processor and peripherals. However, this is not the case in reality: Due to electrical characteristics, peripherals typically operate at lower frequencies, such as SDRAM chips, which usually run at around 100MHz. Higher frequencies can cause timing violations, preventing SDRAM chips from functioning correctly; On the other hand, processors using advanced processes can typically operate at higher frequencies, for example, in a certain version of yzh's multi-cycle NPC, the frequency reaches approximately 1.2GHz under the nangate45 process provided by default in the `yosys-sta` project. Under the above configuration, the SDRAM controller completes one cycle, while the NPC should complete 12 cycles, but Verilator does not detect the frequency difference between the two and simulates them as if they have the same frequency, resulting in simulation results that are much more optimistic than the actual chip, which may also cause some optimization measures to fail to achieve the expected effects on the actual chip.

> **! Update yosys-sta**
>
> We updated the `yosys-sta` project on 2024/04/09 08:30:00, adding a netlist optimization tool developed by the iEDA team, which significantly optimizes the comprehensive netlist generated by yosys, making its timing evaluation results closer to those of commercial tools. If you obtained the `yosys-sta` code prior to the above time, please delete the existing `yosys-sta` project and re-clone it.

To obtain more accurate simulation results to guide us in more effective optimization, we need to calibrate memory access latency. There are two calibration methods: one is to use a simulator that supports multiple clock domains, such as VCS or ICARUS Verilog. Unlike Verilator, which implements a cycle-accurate model, these simulators use an event queue model, treating each computation in Verilog as an event and maintaining event delays, thereby correctly maintaining the order of computations across different modules in multiple

clock domains operating at different frequencies. However, to maintain the event queue model, these simulators typically run slower than Verilator.

The second calibration method involves modifying the RTL code by inserting a delay module into ysyxSOC, which delays requests by a certain number of cycles to simulate the device's operation at low frequencies, ensuring that the number of cycles NPC waits is close to the number it would wait at high frequencies. This method is not overly complex and can be simulated using the faster Verilator, so we opted for this approach. Additionally, this method is also applicable to FPGAs.

Naturally, to implement the delay module, we only need to delay the response to the upstream module after the delay module receives the device's response, rather than responding immediately. However, calculating the number of cycles to wait is a challenge. Considering the example mentioned earlier by yzh, if a request takes 6 cycles in the SDRAM controller, the NPC should wait a total of `6 * 12 = 72` cycles; if the SDRAM controller is refreshing the SDRAM chips and the request takes 10 cycles in the SDRAM controller, the NPC should wait a total of `10 * 12 = 120` cycles; If the request is sent to flash and takes 150 cycles in the SPI master, the NPC should wait a total of `150 * 12 = 1800` cycles. As can be seen, the number of cycles the delay module needs to wait is related to the time the device takes to process the request, and is not a fixed constant, so it needs to be dynamically calculated within the delay module. Assuming the request takes `k` cycles in the device, and the frequency ratio between the processor and the device is `r` (where `r >= 1`), the delay module must calculate the number of cycles the processor needs to wait, `c = k * r`. To perform this dynamic calculation, we need to consider two issues:

1. How to implement multiplication with low overhead?
2. If `r` is a decimal, how to implement decimal multiplication? For example, the frequency reported by the `yosys-sta` project is 550 MHz, so `r = 550 / 100 = 5.5`, but if we calculate 5.5 as 5, a request that takes 6 cycles on the device side will introduce a 3-cycle error on the processor side. For a high-speed CPU, this error is too large, and the accumulation of errors will significantly affect the value of the performance counter, thereby further affecting optimization decisions.

Considering that the ysyxSOC code does not participate in synthesis and tape-out, we can actually use some simple methods to solve the problem, such as using `*` to calculate the result of multiplication and using fixed-point numbers to represent decimals. However, as an exercise, we still require everyone to try a synthesizable implementation method, so that in the future, if you need to solve similar problems in synthesizable circuits, you will know how to do it.

First, let's consider how to implement multiplication when `r` is an integer. Since the delay module itself also needs to wait for the device's response, the waiting time is exactly the number of cycles `k` the request spends in the device. Therefore, we can simply have the delay module increment a counter by `r` each cycle while waiting. Given the processor frequency and device frequency, `r` is a constant value, so it can be directly hard-coded into the RTL code. After the delay module receives the device's response, it enters a waiting state, decrementing the counter by 1 each cycle, and once it reaches 0, it sends the request response back to the upstream.

Next, let's consider the case where `r` is a decimal number. Since the decimal part is inconvenient to handle, simply truncating it would introduce significant error, we can try to incorporate the fractional part into the integer part for accumulation. In fact, we can introduce a scaling factor `s`, adding `r * s` to the counter each cycle during accumulation, so that at the end of accumulation, the counter value is `y = r * s * k`, and before entering the waiting state, the counter is updated to `y / s`. Since `s` is a constant, the result of `r * s` can also be directly hard-coded into the RTL. Of course, `r * s` may not be an integer here, so we truncate it to an integer. Although this theoretically still introduces some error, we can prove that the error is much smaller than before. However, the value of `y` is dynamically calculated and cannot be hard-coded into the RTL. Therefore, for general `s`, `y / s` requires division. You should quickly realize that we can choose some special `s` values to simplify this calculation process! By doing so, we can reduce the error to `1/s` of the original, meaning that the error accumulated during the accumulation phase reaches `s` before the error increases by `1` under this new method.

Looking at the current ysyxSOC, where SDRAM uses an APB interface, we need to implement an APB delay module. The ysyxSOC already includes a framework for an APB delay module, integrated upstream of the APB Xbar, which captures all APB access requests, including those for SDRAM. However, this framework does not provide the specific implementation of the delay module, so it has no delay effect by default. To calibrate the access delay of SDRAM in the ysyxSOC, you also need to implement the functionality of the APB delay module.

Calibrate memory access delay

As described above, implement the APB delay module in ysyxSOC to calibrate the memory access delay in the simulation environment. Specifically, if you choose Verilog, you need to implement the corresponding code in `ysyxSOC/perip/amba/apb_delayer.v`; If you choose Chisel, you need to implement the corresponding code in the `APBDelayerChisel` module in

`ysyxSOC/src/amba/APBDelayer.scala`, and modify `Module(new apb_delayer)` in `ysyxSOC/src/amba/APBDelayer.scala` to instantiate the `APBDelayerChisel` module.

To implement the APB delay module, you need to determine when an APB transaction begins and ends based on the definition of the APB protocol. Assuming an APB transaction starts at time `t0`, the device returns the APB response at time `t1`, and the APB delay module returns the APB response to the upstream at time `t1'`, then the equation `(t1 - t0) * r = t1' - t0` should hold.

Regarding the value of `r`, we assume that the device is running in a 100MHz environment. You can calculate `r` based on the synthesis report from `yosys-sta`. As for `s`, theoretically, the larger the better, but you only need to choose an `s` that is sufficient for practical use. As for how much is sufficient, that is up to you to observe, which is essentially a form of profiling.

After implementation, try different values of `r` and observe in the waveform whether the above equation holds true.

Finding the highest synthesis frequency

In addition to frequency, area is another evaluation metric for circuits. However, at the standard cell level of the process library, these two metrics are inherently mutually constraining: For a class of cells with the same functionality, if you want the cell to have low logic delay, you need to use more transistors to give it stronger drive capability, thereby increasing the cell's area.

Given the trade-off between area and frequency, the synthesizer generally aims to achieve the target frequency with the minimum possible area, rather than considering the maximum clock frequency the circuit can achieve. If your circuit quality is high, you may observe that the frequency in the synthesis report increases as the target frequency is raised, though the synthesis area will also increase accordingly.

Therefore, if you temporarily disregard area constraints, you can set a higher target frequency and allow the synthesizer to attempt to achieve it. In processor design, there are some factors that act as upper limits on processor frequency:

1. Register file read latency. Typically, register file read operations must be completed within a single cycle, and cannot be split across multiple cycles, so the processor's

frequency cannot exceed the maximum operating frequency of the register file.
2. Addition unit latency with bit width and processor word length equal. Typically, addition operations in the EXU must be completed within a single cycle, if this addition operation requires multiple cycles to complete, it will significantly reduce the execution efficiency of all instructions containing addition operations, including addition instructions, subtraction instructions, memory access instructions (requiring calculation of the memory access address), branch instructions (requiring calculation of the target address), or even the calculation of `PC + 4`, thereby significantly reducing the program's IPC. Therefore, the processor's frequency will not exceed the maximum operating frequency of this adder.
3. SRAM read/write latency. As a fully customizable unit, SRAM cannot be optimized for read/write latency through logical design, so as long as SRAM is used, the processor's frequency will not exceed the maximum operating frequency of SRAM.

You can write some simple small modules to evaluate the maximum operating frequency of these components separately, to avoid the influence of I/O ports, you need to insert some flip-flops at the input and output ends of these components. As a fully customizable unit, the maximum operating frequency of SRAM is typically recorded in the corresponding manual, and since we are not currently using SRAM, you can temporarily skip the evaluation of SRAM.

After evaluation, you can set the processor's synthesis target frequency higher than the maximum operating frequencies of the above components, to guide the synthesizer to achieve the highest possible frequency. Of course, you can also directly set the target clock frequency to a value that is difficult to achieve, such as 5000MHz, but we still recommend that you first understand the maximum operating frequencies of these components through the above evaluation process.

Programmable counter increment

The `r` mentioned above is a constant for RTL design, but this is not the case for complex processors with dynamic frequency adjustment. For such complex processors, we need to design `r` as programmable, storing it in a device register, After dynamic frequency adjustment by the software, the new `r` is written to the device register. Therefore, we also need to map this device register to the processor's address space, so that it can be accessed by the processor through the SoC. Of course, `s` can also be designed to be programmable.

However, this requires significant modifications to the ysyxSOC, so we will not require everyone to implement the above programmable functionality.

Re-identify performance bottlenecks

After adding the delay module, rerun some tests and collect performance counter statistics, then use Amdahl's law to identify performance bottlenecks.

Evaluate NPC performance

After adding the delay module, run the microbench train-scale test, record various performance data, including clock frequency information and various performance counters.

After calibrating memory access latency, running the microbench train-scale test in ysyxSOC is expected to take several hours, but we will obtain performance data very close to that of the tape-out environment. Going forward, you can reassess and record performance data after adding each feature, to help you analyze the performance gains brought by each feature.

! Record performance data

Next, we require you to record performance data after each evaluation. If you apply for the tape-out, you will submit this record. If the recorded data does not match the actual development process, you may not be eligible for tape-out. We hope to use this method to force you to recognize and understand the performance changes of NPC, develop the fundamental skills required for processor architecture design, rather than merely translating architecture diagrams from reference books into RTL code.

Specifically, you can record the data as follows:

Commit	Description	Simulation cycles	Instructions	IPC	Sy fr
0123456789abcdef	Example, implement cache	200000	10000	0.05	75

Note:

- We require you to add a rule `make perf` to the `Makefile` in the NPC project directory, so that when you execute

```sh
git checkout the commit in the table
make perf
```

will reproduce the performance data in the corresponding table. If the reproduced results significantly deviate from the data recorded in the table and cannot be reasonably explained, it may be deemed a violation of academic integrity.

- You can assume that you are conducting a scientific research project and are responsible for the experimental data throughout the research process: Experimental data must be reproducible and able to withstand scrutiny in public settings.
- You can create a new worksheet named `NPC Performance Evaluation Results` in your learning records to record these performance data.
- You can replace `Performance Counter 1` and `Performance Counter 2` with the actual names of the corresponding performance counters.
- You can record additional performance counters based on actual circumstances.
- You can also record your analysis of the performance data in the description column.
- We encourage you to record as many performance data entries as possible to help you quantitatively analyze changes in NPC performance.

Is it worth optimizing the CPU clock frequency?

After calibrating the CPU clock frequency to memory access latency, you will find that IPC drops significantly. As expected, if the clock frequency is further increased, the number of cycles for memory access latency will also increase, leading to a decrease in IPC. So, is it worth optimizing the clock frequency? If it is worth it, where exactly does the performance gain from optimizing the clock frequency come from? If it is not worth it, where exactly does the performance degradation from optimizing the clock frequency manifest itself? Try to analyze your hypothesis using performance counters.

💬 Calibrating memory access latency on an FPGA

Modern FPGAs typically include DDR memory controllers. However, due to the implementation principles of FPGAs, the CPU frequency in the PL section differs significantly from that in the ASIC process, and in some cases, the CPU frequency is even lower than that of the memory controller. For example, the memory controller can operate at 200MHz, but the CPU can only run at hundreds of MHz or even tens of MHz on the FPGA, while in real chips, the CPU typically operates at over 1GHz (e.g., the target clock speed for the third-generation Xiangshan is 3GHz). Clearly, performance data obtained in such an evaluation environment is significantly distorted for CPU performance testing targeting tape-out. Addressing the aforementioned memory frequency mismatch through calibration of memory access latency is a problem that processor companies must resolve before using FPGAs for CPU performance evaluation.

In fact, since real DDR is a complex system, even when adopting the latency module solution described above, additional considerations are necessary:

- Due to differences in analog circuit components, the PHY module of the memory controller in the FPGA differs from that of the ASIC's memory controller, which may affect memory access latency.
- Constrained by the implementation principles of FPGAs and the configurable range of PLLs on FPGAs, the operating frequency of the DDR controller is lower than that of the ASIC's memory controller, but DDR memory chips cannot be proportionally downclocked, leading to inaccurate memory access latency.
- After the DDR controller is downclocked, its refresh frequency and other parameters also differ from those of the ASIC memory controller.

Therefore, effectively addressing the memory frequency inversion issue on an FPGA remains a significant challenge in the industry. For example, the Xiangshan team formed a task force led by engineers to tackle this problem.

In fact, whether FPGA memory access latency needs to be calibrated depends on the FPGA's use case and objectives:

- Education: Using the FPGA solely as an environment for functional testing. In this case, the FPGA serves to accelerate the simulation process, and theoretically, the frequency ratio between the processor and memory controller does not affect the results of functional testing.

- Competitions or research projects: Using the FPGA as an environment for performance testing, as well as the target platform. In this case, since the goal is not to tape out, there is no need to calibrate memory access latency.
- Enterprise product development: Using the FPGA as an environment for performance testing, but also aiming to tape out. We expect the performance data obtained from the FPGA to be as consistent as possible with the actual chip. At this point, calibrating the FPGA's memory access latency will be indispensable.

Although “One Student One Chip” has been simplified in many ways, we still hope that everyone can get a general idea of the product development process in a company. At the same time, considering the many engineering challenges involved in calibrating a real DDR controller, we do not require everyone to use an FPGA. In contrast, calibrating memory access latency in a simulation environment is much easier than in an FPGA, so we still recommend that you perform performance evaluation and optimization in a simulation environment.

Improve the efficiency of functional testing

Using the ysyxSoC simulation environment with calibrated memory access latency for performance evaluation is appropriate, but you will also notice that the simulation efficiency of this environment is significantly lower than that of the previous `riscv32e-npc`: From the runtime of the microbench train-scale test, the simulation efficiency of `riscv32e-npc` is tens or even hundreds of times higher than that of `riscv32e-ysyxsoc`. This actually reflects a trade-off: to obtain more accurate performance data, more details need to be simulated (e.g., SDRAM controller and SDRAM chips), which requires more time to simulate one cycle, ultimately leading to lower simulation efficiency. Correspondingly, the `riscv32e-npc` environment with higher simulation efficiency yields inaccurate performance data.

Does this mean `riscv32e-npc` is useless? In fact, we can use `riscv32e-npc` as a functional testing environment. If there is a functional bug in `riscv32e-npc`, it is highly likely that the same bug also exists in `riscv32e-ysyxsoc`, but clearly debugging this bug in the more efficient `riscv32e-npc` is a more appropriate approach. This way, we can leverage the strengths of both simulation environments, complementing each other's weaknesses, thereby improving overall development and testing efficiency.

> Attempt to modify the relevant simulation workflow to support NPC simulation in both `riscv32e-npc` and `riscv32-ysyxsoc`. In this case, `riscv32e-npc` continues to use `0x8000_0000` as the PC value during reset.

Four Optimization Methods for Classic Computer Architectures

After identifying performance bottlenecks, we can consider how to optimize them. Classic computer architectures primarily employ four optimization methods:

1. Locality - Leveraging the nature of data access to enhance instruction and data supply capabilities. Representative technology includes caching.
2. Parallelism - Multiple instances working simultaneously to enhance the system's overall processing capability. Parallel methods further categorize into:
   - Instruction-level parallelism - Execute multiple instructions simultaneously. Related technologies include pipelining, multiple issue, VLIW, and out-of-order execution.
   - Data-level parallelism - Access multiple data simultaneously. Related technologies include SIMD, vector instructions/vector machines.
   - Task-level parallelism - Execute multiple tasks simultaneously. Related technologies include multithreading, multicore, multiprocessing, and multiprocessing; GPUs belong to SIMT, a parallel method that lies between data-level parallelism and task-level parallelism
3. Prediction - When the correct choice is unknown, speculatively execute one choice first, then verify whether the choice was correct in subsequent steps. If the prediction is correct, it reduces waiting latency and improves performance. If the prediction is incorrect, additional mechanisms are required for recovery. Representative technologies include branch prediction and cache prefetching
4. Accelerators - Use specialized hardware components to execute specific tasks, thereby improving the execution efficiency of those tasks. Some examples include:
   - AI accelerators - Accelerate AI workload computations, typically via bus access
   - Custom extended instructions - Integrate accelerators into the CPU and access them via newly added custom extended instructions
   - Multiplier-Divider - Can be considered a type of accelerator, treating RVM as an extension of RVI, controlling dedicated multiplier-divider hardware modules via multiplier-divider instructions to accelerate the computation process

📢 Re-evaluate processor architecture design

Many electronics engineering students may initially misunderstand processor architecture design as “developing a processor using RTL.” However, RTL coding is merely one stage in the processor design process and, strictly speaking, does not fall within the scope of processor architecture design.

In fact, a qualified processor architect should possess the following capabilities:

1. Understand how programs run on processors
2. For features that support program execution, be able to determine whether they are suitable for implementation at the hardware level or the software level
3. For features suitable for implementation at the hardware level, propose a design solution that meets the target requirements after balancing various factors

These capabilities fundamentally reflect the core purpose of using computers: to solve real-world problems through programs. If the benefits of adding features at the hardware level are minimal for a program, or the program does not utilize such features at all, then the decision-makers behind such solutions cannot be considered professional architects.

In fact, these capabilities require deliberate cultivation. We have seen many students who can translate the block diagram of a pipelined processor into RTL code based on reference materials, but are unable to assess whether a program's runtime meets expectations, nor do they know how to further optimize or implement new requirements; Some students design an out-of-order superscalar processor, yet its performance falls short of the five-stage pipeline described in textbooks. This indicates that architectural design capability is not equivalent to RTL coding capability. While these students may have understood the basic concepts of pipelines and out-of-order superscalar processing during development, they lack a holistic perspective and understanding, focusing solely on improving computational efficiency in the backend, paying little to no attention to instruction supply and data supply, resulting in the processor's memory access capability being far below its computational capability, leading to poor overall performance. Therefore, even if one can design a correct out-of-order superscalar processor, it does not qualify as a good processor, and in a sense, these students still lack processor architecture design capabilities.

“One Student, One Chip” aims to cultivate students' processor architecture design capabilities from a different perspective: First, run the program and understand every

detail of its execution from a software-hardware collaboration perspective; Then, learn the basic principles of processor performance evaluation and understand how program behavior manifests at the hardware level; Finally, study various architectural optimization methods and use scientific evaluation tools to understand the actual benefits these optimizations bring to program execution.

This learning approach differs significantly from textbooks, as processor architecture design capabilities can only be developed through practice, but textbook-based theoretical classrooms are constrained by the curriculum system and cannot assess students' computer architecture design capabilities. Therefore, you can easily get started with textbooks or reference books, but if you want to become a professional in this field, you must understand the limitations of these books, and at the necessary stage, go beyond the boundaries of the books and develop true computer architecture design capabilities through targeted training.

Memory Hierarchy and Locality Principle

After calibrating the memory access latency of the ysyxSoC, you should find that the performance bottleneck lies in instruction supply: retrieving a single instruction requires waiting for dozens or even hundreds of cycles, rendering the pipeline ineffective. To enhance instruction supply capabilities, the most suitable approach is to utilize cache technology. However, before introducing caches, we must first understand the computer's memory hierarchy and locality principle.

Memory Hierarchy

Computers utilize various storage media, such as registers, memory, hard disks, and tapes, which possess distinct physical characteristics and thus exhibit differing performance metrics. These can be evaluated based on access time, capacity, and cost.

```
access time     /\        capacity    price
               /  \
   ~1ns       / reg\        ~1KB     $$$$$$
             +------+
   ~10ns    /  DRAM  \      ~10GB     $$$$
           +----------+
   ~10ms  /    disk    \    ~1TB       $$
         +--------------+
```

```
9       ~10s /      tape      \  >10TB       $
10          +------------------+
```

- Registers. Register access time is very short, essentially matching the CPU's clock speed. Current commercial-grade high-performance CPUs have clock speeds of approximately 3GHz, so register access time is less than 1ns. Register capacity is very small, typically less than 1KB. For example, the RV32E has 16 32-bit registers, each 512b in size. Additionally, the manufacturing cost of registers is relatively high, and using a large number of registers will occupy a significant amount of chip area.
- DRAM. DRAM access time is approximately 10 ns, and its capacity is much larger than that of registers, commonly used for memory. Its cost is also much lower; a 16GB memory module on a certain e-commerce platform costs 329 CNY, approximately 20 CNY per GB.
- Mechanical hard drives. The access time of mechanical hard drives is limited by their mechanical components, such as the need for the platters to rotate, typically requiring around 10ms. In contrast, mechanical hard drives also have much larger capacities, typically reaching several TB; their cost is also cheaper, with a 4TB mechanical hard drive priced at 569 CNY on a certain e-commerce platform, approximately 0.139 CNY per GB.
- Solid-state drives (SSDs). SSDs are also a popular storage medium today, using NAND flash memory cells, which operate based on electrical properties, resulting in significantly faster access speeds than mechanical hard drives, with read latency approaching that of DRAM, but write latency remains much higher than DRAM due to the characteristics of flash memory cells. Their cost is slightly higher than that of mechanical hard drives. On a certain e-commerce platform, the price of a 1TB solid-state drive is 699 CNY, approximately 0.683 CNY per GB.
- Magnetic tape. Magnetic tape has a very large storage capacity and very low cost, but its access time is very long, approximately 10 seconds, so it is rarely used today and is typically employed in data backup scenarios. On a certain e-commerce platform, a 30TB tape drive costs 1,000 CNY, approximately 0.033 CNY per GB.

As can be seen, due to the physical limitations of storage media, no single storage device can simultaneously meet all criteria such as large capacity, high speed, and low cost. Therefore, computers typically integrate multiple storage devices and organize them through certain technologies to form a storage hierarchy, achieving a balanced combination of large capacity, high speed, and low cost overall. This may seem somewhat unbelievable, but the key lies in how to effectively organize various storage devices.

Locality Principle

In fact, the above organizational method is carefully designed, and the secret lies in the locality principle of programs. Computer architects have discovered that a program's access to memory over a period of time is typically concentrated within a very small range:

- Temporal locality - After accessing a memory unit, it may be accessed again shortly thereafter
- Spatial locality - After accessing a memory unit, its adjacent memory units may be accessed shortly thereafter

These phenomena are related to the structure and behavior of programs:

- Programs typically execute sequentially or in loops, with sequential execution following spatial locality and loop execution following temporal locality
- When writing programs, related variables are often located close to each other in the source code or organized using structures, and compilers also allocate adjacent memory spaces for them, thereby exhibiting spatial locality
- During program execution, the number of times a variable is accessed is typically no less than the number of variables (otherwise there would be unused variables), so there must be variables that are accessed multiple times, thereby exhibiting temporal locality
- For arrays, programs typically use loops to iterate through them, thereby exhibiting spatial locality

Observing program locality

Program locality is related to memory access, so naturally, we can observe it using mtrace! Run some programs in NEMU and obtain mtrace. Afterward, you need to process the mtrace output further and try to visualize your results using some plotting tools.

Locality of linked lists

Does traversing a linked list exhibit locality? Try comparing the locality of accessing array elements versus linked list elements to determine which exhibits better locality.

The principle of locality tells us that program access to memory exhibits concentrated characteristics. This means that even if slow memory has a large capacity, the program will only access a small portion of the data over a period of time. Given this, we can first move

this portion of data from slow memory to fast memory, and then access it from fast memory.

This is the key to how different types of memory are organized within the memory hierarchy: memories are arranged in a hierarchical structure, with upper-level memories being faster but smaller in capacity, and lower-level memories being larger in capacity but slower; when accessing data, the faster upper-level memory is accessed first; if the data is found in the current level (referred to as a hit), it is directly accessed from the current level; otherwise (referred to as a miss), it is searched for in the next lower level, with the lower level passing the target data and its adjacent data to the upper level. Here, “passing the target data to the upper-level storage” leverages temporal locality, aiming to achieve a hit in the faster storage when accessing the target data again; while “passing adjacent data to the upper-level storage” leverages spatial locality, aiming to achieve a hit in the faster storage when accessing adjacent data again.

For example, when accessing DRAM, if the data is not present, access the mechanical hard drive and move the target data and its adjacent data to DRAM. When accessing these data again, they can be fetched from DRAM, enabling direct access to the data in DRAM. Through this approach, we approximate a memory with access speed close to DRAM and capacity close to a mechanical hard drive! In terms of cost, using the pricing example from the e-commerce platform mentioned earlier, the total cost of a 16GB memory module and a 4TB mechanical hard drive is less than 900 CNY, but if you were to purchase a 4TB memory module, it would cost `329 * (4TB / 16GB) = 84,224` CNY!

Of course, there's no such thing as a free lunch. Achieving the above results comes with conditions: the computer system design must adhere to the principle of locality: On one hand, the computer system must design and implement a storage hierarchy; On the other hand, programmers must develop programs with good locality to achieve optimal performance within the storage hierarchy. If a program has poor locality and the data being accessed lacks centralized characteristics, most accesses will fail to hit in fast memory, resulting in system performance approaching that of accessing slow memory.

Simple Cache

Cache Introduction

Returning to the performance bottleneck mentioned earlier, to optimize instruction supply, what we actually need to do is improve the efficiency of accessing DRAM. To achieve this, we simply need to add an intermediate storage layer between the registers and DRAM,

following the principles of the computer storage hierarchy. This is the concept of a cache (cache). That is, before accessing DRAM, first access the cache; if a hit occurs, access directly; if a miss occurs, first read the data from DRAM into the cache, then access the data in the cache.

The cache mentioned above falls under the narrow definition, referring specifically to the processor cache (CPU cache). In fact, the broader concept of cache extends beyond the hardware module on the memory access path. Caches are ubiquitous in computer systems: disk controllers also include caches to store data read from disks; The row buffer in SDRAM, which we previously discussed, is essentially also a cache for the SDRAM memory array; Operating systems also maintain a cache for storage devices like disks through software, used to store recently accessed data. This cache is essentially a large array of structures, allocated in memory, so the operating system is responsible for moving data between the disk and memory; Caches are also critical for distributed systems. If the data to be accessed is not in the local cache, it must be accessed from a remote location. Browser webpage caches and video content caches fall into this category.

Returning to the CPU cache, who reads data from DRAM into the cache? In fact, who performs the data transfer between the two storage levels depends on who can access both levels. The only entities in a computer are software programs and hardware circuits, and the essence of software programs is a sequence of instructions. Although instructions can access DRAM, the programming model defined in the instruction set manual typically does not include the cache, meaning that, functionally speaking, the cache is invisible to software programs, so it must be the hardware circuits that perform the aforementioned read operation (as we will see later, it is essentially a state machine). For the two storage levels of DRAM and disk, instructions can access DRAM and can also access the disk as a device via MMIO, so software can be responsible for reading data from the disk into DRAM, as mentioned earlier in the operating system. Of course, in principle, you could design a hardware module specifically to handle data transfer between DRAM and disk, but such a hardware module typically only supports one type of disk and lacks the flexibility of a driver in an operating system.

💬 Cache visible to software programs

In some processors, the cache is visible to software programs. For example, shared memory in the CUDA GPU programming model is organized at the same level as CPU cache, serving as an intermediate layer of storage between registers and DRAM; however, unlike CPU cache, the GPU provides specialized memory access instructions

> for accessing shared memory, enabling GPU programs to read data from memory into specific locations in shared memory via instructions.

For convenience, we refer to data read from DRAM as a data block, and data blocks stored in the cache as cache blocks (some textbooks also refer to them as cache lines). Naturally, cache design must address the following issues:

- What should the size of a data block be?
- How should cache hits be determined for memory access requests?
- Since cache capacity is typically smaller than DRAM, how should the mapping relationship between cache blocks and DRAM data blocks be maintained? What happens when the cache is full?
- The CPU may perform write operations to update data in data blocks; how should the cache be maintained in such cases?

Simple Instruction Cache

Let's first consider the instruction cache (ICache). Since the IFU's instruction fetch process does not require writing to memory, the ICache is read-only. We can temporarily disregard how to handle CPU writes to data blocks. Regarding block size, we will initially use the length of a single instruction, which is 4B. This may not be the best design, but for the icache, anything less than 4B is definitely not a good design, as retrieving a new instruction would require multiple memory accesses. Whether a size greater than 4B is better will be evaluated later.

To check if a memory access request is a cache hit, it is natural that, in addition to storing the data block itself, the cache must also record some attributes of the block. The most direct way is to record a unique identifier for the block, but we also want the calculation method for this unique identifier to be sufficiently simple. Since the data block comes from memory, we can number the memory based on block size, and the identifier for the data block corresponding to memory address `addr` is `addr / 4`, which is called the block's tag. Thus, we only need to calculate the tag of the memory access address and compare it with the tags of each cache block to determine whether the target block is in the cache.

Next, we consider the organization of cache blocks. According to the storage hierarchy, the cache capacity cannot be as large as DRAM, and it is also unlikely to be as small as a single cache block. Therefore, we need to consider which cache block to read a new block into when it is read into the cache. Since there are multiple cache blocks in the cache, we can

also assign numbers to the cache blocks. The simplest organization method is to read a new block into a fixed cache block, which is called direct mapping. For this, we need to clearly define the mapping relationship between the memory address `addr` and the cache block number. Assuming the cache can store `k` cache blocks, a simple mapping relationship is `cache block number = (addr / 4) % k`, meaning that for a data block with memory address `addr`, it will be read into the cache block numbered `(addr / 4) % k`.

Obviously, multiple data blocks may map to the same cache block. In this case, we need to decide whether to keep the existing cache block or read the new block into the cache block. According to the principle of program locality, the probability of accessing the new block in the future is higher. Therefore, when reading the new block, we should replace the existing cache block with the new block, so that subsequent accesses to the new block can be cached.

We can view all cache blocks as an array, where the cache block number serves as the array index (index), hence the cache block number is also referred to as the block index. For a direct mapping cache with a block size of `b` bytes and `k` cache blocks, we have `tag = addr / b` and `index = (addr / b) % k`. For convenience in calculation, `b` and `k` are typically chosen as powers of 2, assuming `b = 2^m` and `k = 2^n`. Assuming `addr` is 32 bits, we have `tag = addr / 2^m = addr[31:m]` and `index = (addr / 2^m) % 2^n = addr[m+n-1:m]`. It can be seen that the index is actually the lower `n` bits of the tag. In a direct-mapped cache, data blocks with different indices are always mapped to different cache blocks, even if their upper `m` bits of the tag (i.e., `addr[31:m+n]`) are the same. Therefore, when recording the tag, it is sufficient to record only `addr[31:m+n]`.

A memory access address can be divided into the following three parts: tag, index, and offset. The tag part serves as the unique identifier for the data block in the cache, the index part serves as the index for the data block in the cache, and the offset part is the block-internal offset, indicating which part of the data block needs to be accessed.

```
 31    m+n m+n-1   m m-1    0
+---------+---------+--------+
|   tag   |  index  | offset |
+---------+---------+--------+
```

Finally, when the system is reset, there is no data in the cache, and all cache blocks are invalid. To identify whether a cache block is valid, a validity bit (valid) must be added to each cache block. The valid and tag bits are collectively referred to as the cache block's metadata,

which is data used to manage data, with the data being managed here being the cache block.

In summary, the workflow of the ICache is as follows:

1. The IFU sends an instruction fetch request to the ICache
2. After obtaining the address of the fetch request, the icache uses the index portion to index a cache block, checks whether its tag matches the tag of the requested address, and verifies the validity of the cache block. If all these conditions are met, it is a hit, and the process jumps to step 5.
3. Reads the requested data block from DRAM via the bus.
4. Fills the data block into the corresponding cache block and updates the metadata.
5. Return the fetched instruction to the IFU.

After organizing the workflow, you should know how to implement the icache: it's still a state machine! The above workflow even includes a bus access, so the implementation of the icache can also be seen as an extension of the bus state machine. You are already familiar with the implementation of the bus, so how to implement the icache state machine is up to you to figure out!

Implement the icache

Based on the above workflow, implement a simple icache with a block size of 4B and a total of 16 cache blocks. Generally, the cache storage array (including data and metadata) is implemented using SRAM, but using SRAM in the ASIC process involves selection and instantiation, where the selection of SRAM may affect the storage of data and metadata. As the first cache exercise, for simplicity, the storage array is first implemented using flip-flops to enhance implementation flexibility.

During implementation, it is recommended to design relevant parameters as configurable to facilitate subsequent performance evaluation of different configuration parameters. After implementation, attempt to evaluate its performance.

Address spaces suitable for caching

Not all address spaces are suitable for caching; only memory-type address spaces are suitable for caching. Additionally, SRAM access latency is only 1 cycle, so caching

is unnecessary. Allocating cache blocks to other address spaces is a more appropriate solution.

Estimating the ideal performance gain of the dcache

Typically, LSUs are paired with a cache called the data cache (dcache). Before implementing the dcache, we can first estimate its performance gain under ideal conditions. Assuming the dcache has infinite capacity, a 100% hit rate when accessing the dcache, and a latency of 1 cycle, attempt to estimate the performance benefit of adding such a dcache based on performance counters.

If your estimate is correct, you should find that adding the dcache is not worthwhile at this point. We will continue to discuss this issue in the following sections.

Formal Verification

With DiffTest, you should be able to easily ensure that the given program still runs correctly after integrating icache. But how can we ensure that icache runs correctly for any program?

This seems like a difficult problem, and I'm sure you've encountered this situation before: the code runs correctly for the given test cases, but one day when running another test, it fails. Whether analyzed from a theoretical perspective or summarized from practical experience, testing alone cannot prove the correctness of a module, unless the test cases cover all programs or all input scenarios of the tested module. The number of programs is infinite, so testing all programs is impractical, but the inputs of the tested module are finite, so at least iterating through all inputs is theoretically feasible.

To traverse all inputs of a module, one must generate a test set that covers all input scenarios, and also have a method to determine whether a specific input is correct. Even if these are achievable, running all tests would take a significant amount of time, which is often intolerable. The equivalence class testing method in software testing theory can categorize tests with similar essential behaviors, select one test from the equivalence class to represent the entire equivalence class, thereby reducing the size of the test set. However, how to divide equivalence classes requires manual decision-making based on the logic of the tested module. However, according to another widely circulated piece of advice in the software field, any process requiring manual intervention carries the risk of errors.

Basic Principles of Formal Verification

Can tools help us automatically find test cases? There are indeed such tools! Solvers are mathematical tools that find feasible solutions under given constraints, similar in essence to solving systems of equations or linear programming problems. For example, Z3 is a solver for Satisfiability Modulo Theories (SMT) problems, which can determine whether a proposition containing real numbers, integers, bits, characters, arrays, strings, etc., is valid. In fact, as long as a problem can be expressed as a subset of first-order logic, it can be solved by an SMT solver, so SMT solvers can also be used to solve complex problems like Sudoku. SMT solvers are widely used in fields such as automated theorem proving, program analysis, program verification, and software testing. Below is an example of using Z3 to solve a system of equations in Python.

```py
#!/usr/bin/python
from z3 import *

x = Real('x')  # Define variables
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)    # Define constraints
s.add(2*x - 2*y - 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print(s.check())  # Check if a feasible solution exists: sat
print(s.model())  # Output feasible solution: [y = 14/25, x = 1/25,
z = 6/25]
```

In the field of testing and verification, there is a verification method called formal verification. One technical approach to this method is model checking. The corresponding verification tool is called a model checker. The core of model checking is the solver. Specifically, the model checker treats the design as constraints, the input as variables, and "at least one verification condition is not satisfied" as the solution objective. These elements are expressed using first-order logic and converted into a language recognizable by the solver, which then attempts to find a feasible solution. For example, if a design contains the verification conditions `assert(cond1)` and `assert(cond2)`, the solver attempts to find an input such that `!cond1 || !cond2` holds. If a feasible solution exists, it means the solver has found a test case that violates the verification condition, and this counterexample can

help us debug and improve the design; if no feasible solution exists, it means no input will violate the verification condition, thereby proving the correctness of the design! As can be seen, whether or not the solver finds a feasible solution, it is excellent news for the designer.

Generally speaking, the state of a system is also related to time. For example, the state of a processor may change after each cycle. If a condition can be verified at all time intervals, the method is called unbounded model checking (UMC). However, due to the high computational cost of UMC, bounded model checking (BMC) is more commonly used in practice. BMC typically requires a parameter `k`, which only verifies whether a condition holds after a maximum of `k` time units. This parameter is called the bound of BMC.

📢 Do not blindly trust UVM test 100% coverage reports.

If you understand UVM, you should know that its goal is to improve coverage. However, if you believe that improving coverage is the ultimate goal of test verification, then you likely do not fully understand test verification.

In fact, the ultimate goal of test verification is to prove the correctness of the design or to identify all bugs in the design. But experienced engineers know that even with 100% coverage, there may still be some bugs that are not detected, and it is impossible to estimate how many of these undetected bugs there are.

The reason why coverage targets are widely adopted by companies is, on the one hand, because coverage is an easy-to-quantify and statistically measurable metric. If we describe “an event being covered” in strict terms, it is

```
There exists a test case that runs successfully and triggers
the event during execution.
```

Here, the event can be executing a specific line of code (line coverage), a signal flipping (flip coverage), a state transition in a state machine (state machine coverage), or a custom condition being met (functional coverage), etc. Meanwhile, “coverage reaching 100%” means

```
For each event, there exists a test case that runs
successfully and triggers the event during execution.
```

Note that we can use different test cases to cover different events. According to this definition, we only need to add some flags during simulation to calculate the

coverage. Most RTL simulators (including Verilator) even provide automatic coverage statistics functionality. If you want to learn how to calculate coverage, simply RTFM.

On the other hand, as can be seen from the above definition, improving coverage is actually the lower limit of verification work. If coverage is too low, it only indicates that verification work is insufficient, which aligns with the principle that “untested code is always wrong.” However, the ultimate goal of testing and verification is

```
For all test cases, they all run successfully.
```

In contrast, “coverage reaching 100%” is actually a necessary but insufficient condition for “correct design,” and it is very lenient. In fact, we can easily provide a counterexample: A module has two functions, and each function is covered by its own test case, resulting in 100% functional coverage; However, when running test cases that require interaction between the two functions, errors occur.

Compared to verification work with low coverage, achieving higher coverage certainly increases the probability of correct design, but what we want to emphasize is that even with 100% coverage, it is still far from sufficient. Especially for complex systems, some deeply hidden bugs typically only trigger when multiple boundary conditions are simultaneously met. Rather than insisting on 100% coverage as the verification target, we encourage everyone to actively explore other methods and techniques to identify more potential bugs, which holds greater practical significance for verification work.

A simple example of formal verification

Formal verification process based on Chisel

Chisel's testing framework **chiseltest** has integrated formal verification functionality, which can translate FIRRTL code into a language recognizable by BMC and allow BMC to prove whether the given `assert()` is correct. If a counterexample is found, the waveform generating that counterexample is generated to assist in debugging, which is very convenient. With formal verification tools, we no longer need to worry about incomplete test case coverage, and we may not even need to write test cases at all. In a word, it's fantastic!

Below is an example of formal verification of a Chisel module:

```scala
import chisel3._
import chisel3.util._
import chiseltest._
import chiseltest.formal._
import org.scalatest.flatspec.AnyFlatSpec

class Sub extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val c = Output(UInt(4.W))
  })
  io.c := io.a + ~io.b + Mux(io.a === 2.U, 0.U, 1.U)

  val ref = io.a - io.b
  assert(io.c === ref)
}

class FormalTest extends AnyFlatSpec with ChiselScalatestTester with
Formal {
  “Test” should “pass” in {
    verify(new Sub, Seq(BoundedCheck(1), BtormcEngineAnnotation))
  }
}
```

> **! No longer use Utest**
>
> With the evolution of Chisel versions, Utest is no longer supported, so we also recommend that you no longer use Utest. If you obtained the `chisel-playground` code before 2024/04/11 01:00:00, please refer to the `object test` section in the **new version of** `build.sc` to modify your `build.sc` .

The `Sub` module in the above example implements two's complement subtraction using “invert and add 1.” To verify the correctness of the `Sub` module implementation, the code compares the result of “invert and add 1” with the result obtained using the subtraction operator. We expect `assert()` to hold true for any input. To demonstrate the effectiveness of formal verification, we injected a bug into the implementation of the `Sub` module: when `io.a` is `2` , the “add 1” operation is not performed, resulting in an incorrect two's complement subtraction result.

When calling the formal verification functionality provided by chiseltest, the above code also needs to pass in a `BoundedCheck(1)` parameter as the BMC boundary, to specify the number of cycles to be proven. For example, `BoundedCheck(4)` indicates that BMC should attempt to prove that the tested module does not violate `assert()` within 4 cycles after reset, under any input signal. For combinational logic circuits, we only need BMC to solve within 1 cycle.

Additionally, the above code passes a `BtormcEngineAnnotation` parameter, indicating the invocation of the model detector **BtorMC** **btormc**. BtorMC is based on an SMT solver **Boolector** that is different from Z3. Its basic principle is similar to Z3, but according to actual testing, its solving efficiency is typically several times that of Z3. Before running the above test, you also need to obtain the BtorMC tool. Specifically, you need to download the corresponding tool from **this link**. After unzipping, add `path-to-oss-cad-suite/bin` to the environment variable `PATH` to invoke BtorMC.

After completing the above configuration, run the test using `mill -i __.test`, and the output will be as follows:

```
Assertion failed
    at SubTest.scala:16 assert(io.c === ref)
- should pass *** FAILED ***
  chiseltest.formal.FailedBoundedCheckException: [Sub] found an
assertion violation 0 steps after reset!
  at
chiseltest.formal.FailedBoundedCheckException$.apply(Formal.scala:26
)
  at chiseltest.formal.backends.Maltese$.bmc(Maltese.scala:92)
  at chiseltest.formal.Formal$.executeOp(Formal.scala:81)
  at chiseltest.formal.Formal$.$anonfun$verify$2(Formal.scala:61)
  at
chiseltest.formal.Formal$.$anonfun$verify$2$adapted(Formal.scala:61)
  at scala.collection.immutable.List.foreach(List.scala:333)
  at chiseltest.formal.Formal$.verify(Formal.scala:61)
  at chiseltest.formal.Formal.verify(Formal.scala:34)
  at chiseltest.formal.Formal.verify$(Formal.scala:32)
  at FormalTest.verify(SubTest.scala:19)
  ...
```

The above information indicates that BMC found a test case that violates `assert()` in the 0th cycle after reset. Furthermore, developers can use the waveform file `test_and_run/Test_should_pass/Sub.vcd` to assist in debugging. After correcting the error in the `Sub` module, rerunning the above test will no longer output error messages,

indicating that BMC cannot find counterexamples, thereby proving the correctness of the code.

Formal verification process based on Verilog

The formal verification process of chiseltest converts FIRRTL code into a language that can be recognized by BMC. It does not involve Verilog, so the above process does not support projects developed based on Verilog. If you are developing using Verilog, you can use the formal verification process based on Yosys, where SymbiYosys is the front-end tool for this process.

The following is an example of formal verification for a Verilog module:

```
// Sub.sv
`define FORMAL

module Sub(
  input [3:0] a,
  input [3:0] b,
  output [3:0] c
);

  assign c = a + ~b + (a == 4'd2 ? 1'b0 : 1'b1);

`ifdef FORMAL
  always @(*) begin
    c_assert: assert(c == a - b);
  end
`endif  // FORMAL

endmodule
```

The `Sub` module in the above example implements two's complement subtraction using “invert and add 1”. To verify the correctness of the `Sub` module's implementation, the code compares the result of “invert and add 1” with the result obtained using the subtraction operator. We expect `assert()` to hold true for any input. To demonstrate the effectiveness of formal verification, we have introduced a bug into the implementation of the `Sub` module: When `a` is `2`, the “add 1” operation is not performed, resulting in an incorrect two's complement subtraction result.

After writing the above `Sub.sv` file, you also need to write the SymbiYosys configuration file `*.sby`, which generally consists of the following parts:

- task: optional, used to specify the tasks to be executed
- options: required, used to map statements such as `assert` and `cover` in the code to the model
- engines: required, used to specify the model to be solved
- script: Required, contains the Yosys script needed for testing
- files: Required, used to specify the files for testing

The following is an example of the configuration file `Sub.sby`:

```
[tasks]
basic bmc
basic: default

[options]
bmc:
mode bmc
depth 1

[engines]
smtbmc

[script]
read -formal Sub.sv
prep -top Sub

[files]
Sub.sv
```

The `depth` option in the above configuration is the boundary of BMC, used to specify the number of cycles to be proven. For example, `depth 4` means that BMC will attempt to prove that the tested module does not violate `assert()` in any input signal within 4 cycles after reset. For combinational logic circuits, we only need BMC to solve within 1 cycle.

Before performing formal verification, you need to download the corresponding tools from **this link**. After unzipping, execute the command line `path-to-oss-cad-suite/bin/sby -f Sub.sby` to perform formal verification, and the output information is as follows:

```
SBY 16:52:19 [Sub_basic] engine_0: ##   0:00:00  Checking
assumptions in step 0..
```

```
SBY 16:52:19 [Sub_basic] engine_0: ##   0:00:00  Checking assertions in step 0..
SBY 16:52:19 [Sub_basic] engine_0: ##   0:00:00  BMC failed!
SBY 16:52:19 [Sub_basic] engine_0: ##   0:00:00  Assert failed in Sub: c_assert
SBY 16:52:19 [Sub_basic] engine_0: Status returned by engine: FAIL
SBY 16:52:19 [Sub_basic] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 16:52:19 [Sub_basic] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 16:52:19 [Sub_basic] summary: engine_0 (smtbmc) returned FAIL
SBY 16:52:19 [Sub_basic] summary: counterexample trace: Sub_basic/engine_0/trace.vcd
SBY 16:52:19 [Sub_basic] summary:   failed assertion Sub.c_assert at Sub.sv:11.9-11.37 in step 0
SBY 16:52:19 [Sub_basic] DONE (FAIL, rc=2)
SBY 16:52:19 The following tasks failed: ['basic']
```

The above information indicates that BMC found a test case that violated `assert()` in the 0th cycle after reset. Furthermore, developers can use the waveform file `Sub_basic/engine_0/trace.vcd` to assist in debugging. After correcting the error in the `Sub` module, rerunning the above command will output a success message, indicating that BMC cannot find a counterexample, thereby proving the correctness of the code.

Verifying the icache through formal verification

Our goal is to prove the correctness of the icache through formal verification, so we first need to design the corresponding REF and determine the verification conditions. Since the cache is a technology designed to improve memory access efficiency, it should not affect the correctness of memory access results, i.e., the behavior of memory access requests should be consistent regardless of whether the cache is present. Therefore, we can use the simplest memory access system as the REF, which receives memory access requests from the CPU and directly accesses memory; the corresponding DUT, on the other hand, routes memory access requests through the cache. For the verification conditions, we only need to check whether the results returned by read requests are consistent.

Based on the above analysis, we can easily write pseudocode for the verification top-level module. Here we use Chisel as pseudocode, but if you are developing in Verilog, you can still draw on the relevant ideas to write the verification of the top-level module.

```
class CacheTest extends Module {
  val io = IO(new Bundle {
    val req = new ...
    val block = Input(Bool())
  })

  val memSize = 128  // byte
  val mem = Mem(memSize / 4, UInt(32.W))
  val dut = Module(new Cache)

  dut.io.req <> io.req

  val dutData = dut.io.rdata
  val refRData = mem(io.req.addr)
  when (dut.io.resp.valid) {
    assert(dutData === refData)
  }
}
```

The above pseudocode only provides a rough framework. You need to add some details based on your specific implementation:

- Block write operations by setting the write enable signal to `0`
- When the cache is missing, read data from `mem`. Since the test object does not generate write operations, the DUT and REF can use the same memory
- Since REF reads data directly from `mem` with no delay, while DUT reads data from the cache and needs to go through several cycles, the timing of `assert()` needs to be synchronized: After REF reads the data, it must wait for the DUT to return the read result before performing the check. This can be easily achieved using a state machine.
- Since formal verification tools traverse all input conditions for each cycle, the input signals change every cycle. You may need to use registers to temporarily store some results.
- Leveraging the feature that "formal verification tools iterate through all input conditions for each cycle," we can define some `block` signals at the top level of the test, to verify whether AXI-related code can function under random delay scenarios, for example, `dut.io.axi.ar.ready := aready_ok & ~block1`, `dut.io.axi.r.valid := rvalid_ok & ~block2`

> **Verify the implementation of icache through formal verification**

> Although this is not mandatory, we strongly recommend that you try this modern testing and verification method, and experience the satisfaction of "solving problems with the right tools." Regarding the boundary of BMC (`BoundedCheck()` or `depth`), you can select an appropriate parameter to ensure that the cache can process 3–4 requests within the cycle proven by the formal verification tool, thereby testing whether the cache can correctly handle any consecutive requests.

Formal verification seems to have only advantages, but it actually has a fatal flaw: the state space explosion problem. As the design scale grows and the bounds increase, the space that the solver needs to traverse also grows. In fact, first-order logic languages are theoretically undecidable; even decidable subsets are typically NP-Hard problems in terms of algorithmic complexity. This means that the solver's runtime is likely to grow exponentially with the design scale. Therefore, formal verification is generally used in unit testing.

Cache Optimization

Since cache technology is primarily used to improve memory access efficiency, it is natural to evaluate cache performance using memory access-related metrics. Typically, AMAT (Average Memory Access Time) is used to assess cache performance, assuming a cache hit rate of `p` :

```
AMAT = p * access_time + (1 - p) * (access_time + miss_penalty)
     = access_time + (1 - p) * miss_penalty
```

where `access_time` is the cache access time, i.e., the time required from receiving a memory access request to obtaining a hit result, and `miss_penalty` is the cost of a cache miss, which in this case is the time required to access DRAM.

This equation provides guidance for optimizing cache performance: reduce access time `access_time` , increase hit rate `p` , or reduce miss penalty `miss_penalty` . In the current NPC, there is limited room for optimization of access time in architectural design, as it is more influenced by specific implementations, such as cycle count and critical path. Therefore, we will focus on optimizing hit rate and miss penalty in the following sections.

> **Statistics on AMAT**

Add appropriate performance counters to the NPC to track the AMAT of the icache.

Optimizing Hit Rate

To improve the hit rate, we need to reduce the miss rate. To do this, we first need to understand the causes of cache misses.

The 3C Model of Cache Misses

Computer scientist Mark Hill proposed the 3C model in his 1987 PhD thesis, which describes three types of cache misses:

1. Compulsory miss, defined as a miss that occurs in a cache with infinite capacity, manifested as a miss during the first access to a data block
2. Capacity miss, defined as a miss that cannot be eliminated without expanding the cache capacity, manifested as a miss occurring because the cache cannot accommodate all the required data.
3. Conflict miss, defined as a miss caused by reasons other than the above two, manifested as a miss occurring due to the replacement of multiple cache blocks.

With the 3C model, we can propose targeted solutions for each type of miss to reduce the corresponding miss rate.

Reducing Compulsory Misses

To reduce compulsory misses, in principle, a data block should be read into the cache before it is accessed. However, the aforementioned cache workflow does not support this functionality, so a new mechanism must be added to achieve it. This mechanism is called prefetching (prefetch). However, theoretically, Compulsory miss only refers to the miss that occurs during the first access to a data block. In scenarios with a high number of memory accesses, the proportion of Compulsory miss is not high. Therefore, we will not delve into how to reduce Compulsory miss here. Interested readers can search for and read relevant materials on prefetch.

Reducing Capacity Misses

To reduce capacity misses, according to its definition, the only option is to increase cache capacity to better utilize temporal locality. However, larger cache capacity is not always

better. On one hand, larger cache capacity means larger chip area, thereby increasing chip production costs; on the other hand, larger storage arrays mean higher access latency, thereby increasing cache access time and reducing cache performance. Therefore, in actual projects, blindly increasing cache capacity is not a reasonable solution; a balanced decision must be made after considering all factors.

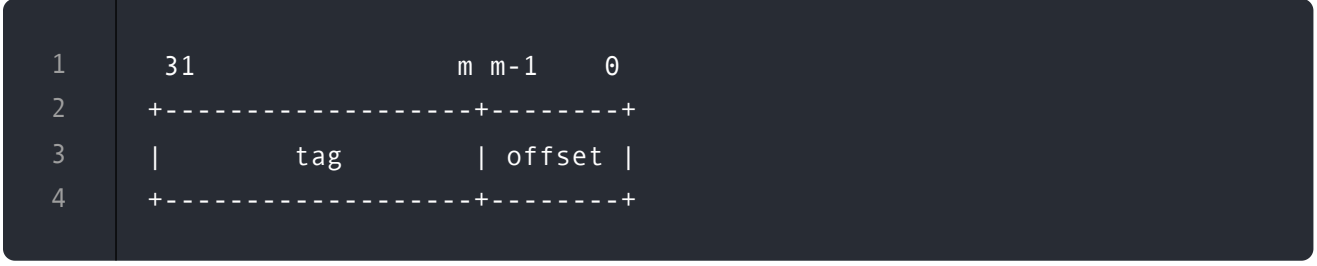
Reducing Conflict Misses

To reduce conflict misses, it is necessary to consider how to minimize the replacement of multiple cache blocks. As mentioned earlier, the direct mapping organization in the cache means that each data block can only be read into a cache block with a fixed index. If multiple data blocks have the same index, the later-read data block will replace the earlier-read data block. Therefore, one approach to reduce replacements is to adopt a new cache block organization scheme that allows data blocks to be read into multiple cache blocks.

An extreme case is where each data block can be stored in any cache block, which is referred to as a fully-associative organization scheme. The specific cache block to store in is first determined by selecting an invalid cache block; if all cache blocks are valid, the replacement algorithm determines the destination. Different replacement algorithms affect cache access hit rates, generally, the replacement algorithm should select the cache block least likely to be accessed in the future. For a given memory access sequence, we can design an optimal replacement algorithm to minimize conflict misses; However, in practice, we cannot predict future memory access sequences in advance, so the design of the replacement algorithm becomes a problem of “predicting the future based on the past,” i.e., predicting the cache block that is least likely to be accessed in the future based on the access history of each cache block. Common replacement algorithms include the following:

- FIFO (First-In, First-Out): Replace the oldest read cache block
- LRU (Least Recently Used): Replaces the cache block with the fewest accesses in the recent period.
- Random: Replaces cache blocks randomly.

When combined with an appropriate replacement algorithm, the fully associative organization can replace the cache block least likely to be accessed in the near future with a higher probability, thereby minimizing conflict misses to the greatest extent possible. However, since the fully associative organization can store data blocks in any cache block, this incurs two costs. First, the memory access address does not need to be divided into an index part, so except for the offset part, the rest are tag parts. Therefore, we need to spend more storage overhead in the storage array to store the tag part of the cache block.



Second, when determining a hit, it is necessary to check whether the tags of all cache blocks match, which requires the use of many comparators, thereby increasing area overhead. Due to these costs, the fully associative organization method is generally only used in scenarios with a small number of cache blocks.

Set-associative is a compromise between direct mapping and fully associative. The idea is to group all cache blocks, select one group among the groups using direct mapping, and then select one cache block within the group using fully associative mapping. That is, each data block can be stored in any cache block with a group number of `tag % number of groups`. If each group contains `w` cache blocks, it is referred to as `w-way set-associative`.

```
                  v
               hit way
```

In the set associative organization method, a memory access address can be divided into the following three parts: tag, index, and offset. The index part is used as the set index, so its bit width is `n = log2(total number of cache blocks/w)`.

```
 31    m+n m+n-1   m m-1    0
+---------+---------+--------+
|   tag   |  index  | offset |
+---------+---------+--------+
```

When determining a hit, it is only necessary to check whether the tag matches all cache blocks within the group. When `w` is not large, the area overhead of the comparator is acceptable.

In fact, fully associative and direct mapping can both be regarded as special cases of set associative: when `w=1`, it is direct mapping; when `w=total number of cache blocks`, it is fully associative. Modern CPUs typically use 8 or 16-way set associative.

Block Size Selection

Block size is a special parameter. If the cache block is larger, on one hand, it reduces the storage overhead of tags, on the other hand, it can store more adjacent data within the cache block, thereby better capturing the spatial locality of the program and reducing conflict misses. However, to read more adjacent data, the cost of cache misses also increases; Additionally, for a given cache capacity, larger cache blocks mean fewer cache blocks, and if the program's spatial locality is not pronounced, more smaller cache blocks are needed, in which case larger cache blocks may actually increase conflict misses.

```
// Program with poor spatial locality
// 2 cache blocks of size 4
               1111       2222        cache
|------------ -oooo------oooo-----|  memory, `o` represents the
program's hotspot data

// 1 cache block of size 8
               11111111               cache
|--------------oooo------oooo-----|  memory

```

```
11
12
13  // Program with good spatial locality
14  // 2 cache blocks of size 4
15                 11112222                cache
16  |--------------oooooooo------------|   memory
17
18  // 1 cache block of size 8
19                 11111111                cache
    |--------------oooooooo------------|   memory
```

Design Space Exploration

The above text mentions many cache-related parameters. How to select a suitable set of parameters to achieve good performance under given resources belongs to the cache design space exploration (DSE) problem. Currently, the performance metrics we are concerned with include IPC, clock frequency, and area. Among these, clock frequency and area can be quickly evaluated using the `yosys-sta` project, while IPC typically requires running the entire program in a ysyxSoC environment calibrated for memory access latency to obtain.

However, the number of possible parameter combinations is too large. If we were to spend hours evaluating each combination, the efficiency of design space exploration would be extremely low. As mentioned earlier, data accuracy and simulation efficiency are trade-offs. Therefore, to improve the efficiency of design space exploration, one approach is to sacrifice the statistical accuracy of IPC, and instead use a metric that reflects the trend of IPC changes with lower computational overhead.

When adjusting various cache parameters, the direct impact is on AMAT, so we can assume that the execution overhead of other parts of the CPU remains constant. According to the definition of AMAT, adjusting the aforementioned parameters does not affect cache access time, so it can be treated as a constant. Therefore, what we truly need to focus on is the total time the program must wait when a cache miss occurs, which we refer to as the total miss time (TMT). In fact, TMT can represent the trend of IPC changes: the smaller the TMT, the fewer cycles required per instruction for memory access, resulting in a higher IPC.

When you previously used performance counters to measure AMAT, you should have also measured TMT, but this requires running the program in ysyxSoC. To measure TMT with low overhead, we consider the angle of “miss count * miss cost,” and explore how to measure miss count and miss cost with low overhead.

For counting miss counts, we have the following observations:

- For a given program, the number of cache accesses is fixed. By obtaining the program's execution trace (itrace) and inputting it into the icache, we can simulate the icache's operation process, thereby counting the icache's miss counts, so there is no need to simulate the entire ysyxSoC, or even the NPC.
- When the NPC executes a program, it needs to determine the next instruction to execute based on the execution result of the current instruction. However, the itrace already contains the complete instruction stream, so when counting TMT, we only need the PC value of the instruction stream, not the instruction itself.
- The data portion of the icache is used as the IFU to be returned to the NPC, but since TMT counting does not require the NPC, the data portion of the icache is also unnecessary, and only the metadata portion needs to be retained. In fact, for a given memory access address sequence, the number of cache misses is independent of the memory access content, and the correct number of misses can be counted by maintaining the metadata.

Therefore, to count the number of misses in the icache, there is no need to run the program in full each time. What we truly need is a simple cache functionality simulator, which we refer to as cachesim. Cachesim receives the PC sequence of the instruction stream (a simplified version of itrace) and counts the number of misses for this PC sequence by maintaining metadata. As for the PC sequence of the instruction stream, we can quickly generate it using NEMU.

As for the miss cost, since cachesim does not include memory access details from ysyxSoC, we cannot accurately obtain this value in principle. However, as discussed earlier, only the block size parameter affects the miss cost, so we can calculate an average miss cost in ysyxSoC and use it as a constant to estimate TMT.

Implement cachesim

Based on the above introduction, implement a simple cache simulator.

Using cachesim, we can perform performance testing DiffTest on the icache. Specifically, we can use cachesim as the performance testing REF, and the performance counter results obtained by running a program in NPC should be completely consistent with the miss count data and other statistics obtained by cachesim executing the corresponding PC sequence. If there is a discrepancy, it may indicate a performance bug in the RTL implementation, which cannot be detected through functional testing with NEMU or formal verification. For example, even if the iCache is consistently missing, the program may still run correctly on the NPC. Of

course, it is also possible that cachesim, which is used as a REFERENCE, has a bug. However, regardless of the cause, having a REFERENCE for comparison is always beneficial.

However, to obtain consistent itrace results, you may need to make some modifications to NEMU to enable it to run the `riscv32e-ysyxsoc` ISO file.

Compressing the trace

If the itrace you obtain is very large, you can consider compressing it in the following ways:

- Store the itrace in binary format instead of text format
- Most of the time, instructions are executed sequentially. For a continuous PC sequence, we can only record the first PC and the number of continuously executed instructions
- Further compress the generated itrace file using `bzip2` -related tools, then obtain a readable file pointer in the cachesim code via `popen("bzcat compressed file path", "r")` . For information on how to use `popen()` , please RTFM

Using cachesim for design space exploration

With cachesim, we can quickly evaluate the expected benefits of different cache parameter combinations. For a given parameter combination, cachesim's evaluation efficiency is thousands or even tens of thousands of times faster than ysyxSoC.

Additionally, we can utilize multiple cores to evaluate multiple parameter combinations simultaneously: Specifically, first pass various cache parameters to cachesim via the command line, then use a script to launch multiple instances of cachesim, each with different parameter combinations. Through this approach, we can obtain evaluation results for dozens of parameter combinations within minutes, thereby helping us quickly select suitable parameter combinations.

Attempt to establish this rapid evaluation workflow and assess several parameter combinations. However, we have not yet optimized the miss cost to evaluate a more reasonable TMT; additionally, we have not yet imposed any area size constraints.

These factors will influence the final decision, so you do not need to make a final design choice at this stage.

Optimizing Miss Cost

If a cache miss occurs, data must be accessed from the next storage layer, so the miss cost is the access time of the next storage layer. There are many techniques to reduce the miss cost, and we will first discuss one of them: burst access over the bus. If the cache block size matches the bus data width, only a single bus transaction is needed to access the data block, leaving little room for optimization. However, for larger cache blocks, multiple bus transactions are theoretically required to access the data block, presenting opportunities for optimization.

Block Size and Miss Cost

For the current cache design, the next storage layer is SDRAM. To further analyze this, we first establish the following simple model for SDRAM access time: SDRAM access time is divided into four segments, so for an independent bus transfer transaction, the overhead is `a+b+c+d`. Assuming the bus data width is 4 bytes and the cache block size is 16 bytes, if four independent bus transfer transactions are used, the required overhead is `4(a+b+c+d)`.

```
+------------------------ arvalid valid
|   +------------ -------- AR channel handshake, receive read
request
|   |       +------------ State machine transitions to READ state
and sends READ command to SDRAM die
|   |       |     +------ SDRAM die returns read data
|   |       |     |   +-- R channel handshake, return read data
V a V   b   V  c  V d V
|---|-------|-----|---|
```

Support larger block sizes

Modify the block size in cachesim to be four times the bus data width, and estimate the missing cost using an independent bus transaction method.

> After implementation, compare with previous evaluation results and attempt to analyze the causes.

Bus burst transfers

Similar to SDRAM chips, the AXI bus also supports “burst transfers,” which involve multiple consecutive data transfers within a single bus transaction, with each data transfer referred to as a “beat.” In the AXI bus protocol, the `arbusrt` signal on the `AR` channel is used to indicate whether the transfer is a burst transfer. If so, the `arlen` signal is also used to indicate the number of beats in the burst transfer.

Of course, bus protocol support alone is insufficient; the AXI master must initiate the burst transfer transaction, and the slave must support processing of burst transfer transactions. As the master end, we will leave the initiation of burst transfer transactions as an experimental task for you to explore. On the other hand, the SDRAM controller provided by ysyxSoC does support the processing of burst transfer transactions, allowing the aforementioned four bus transfers to be included in a single bus transaction, thereby effectively reducing the overhead of fully reading a data block: First, through burst transfer, the `AR` channel only needs to perform a single handshake, saving `3a` in overhead compared to the previous scheme; Second, the SDRAM controller can split a single burst transfer transaction into multiple READ commands sent to the SDRAM chips. When the SDRAM chips return a read data, if there are still READ commands with consecutive addresses, the state machine of the SDRAM controller directly transitions to the READ state to continue sending READ commands, saving `3b` in overhead compared to the previous scheme; Finally, the SDRAM controller can respond to the `R` channel while sending the next READ command, thus hiding the overhead of the `R` channel handshake. Compared to the previous scheme, this saves `3d` in overhead.

```
  a     b      c    d
|---|-------|-----|---| <-------------------- 1st beat
                  |-----|---| <--------------- 2rd beat
                        |-----|---| <--------- 3rd beat
                              |-----|---| <-- 4th beat
```

In summary, the overhead required for burst transmission is `a+b+4c+d`, which saves `3(a+b+d)` compared to the previous scheme. Extending this to a general case, if the cache block size is `n` times the bus data width, burst transmission can save `n(a+b+d)`

overhead. Although `a`, `b`, and `d` appear small at first glance, remember that this is the overhead from the SDRAM controller's perspective. After calibrating the memory access latency, this could save dozens or even hundreds of cycles for the CPU.

Implementation of Burst Transmission

As can be seen, to achieve the benefits of burst transmission, the cache must first use larger blocks. However, as discussed above, larger cache blocks may also increase conflict misses, which may result in negative benefits, depending on the spatial locality of the program. To determine which is better, benchmarking is required. To evaluate TMT in cachesim, we also need to obtain the missing cost when using burst transmission. To do this, we first need to implement burst transmission in the ysyxSoC environment. However, this change involves many details, so we will proceed in several steps.

Previously, for testing convenience, we first integrated an SDRAM controller with an APB interface. However, the APB protocol does not support burst transfers. To adopt burst transfers, we first need to replace the SDRAM controller with an AXI interface version.

! Switch to 32-bit ysyxSoC

The SoC team has provided a 32-bit production SoC. We also updated `ysyxSoC` to 32-bit on 2024/07/26 13:00:00, to help everyone test locally before integrating the production SoC. If you obtained the `ysyxSoC` code before the aforementioned time, please refer to the content at the beginning of this page for instructions.

Integrate the AXI interface for the SDRAM controller

You need to modify the `sdramUseAXI` variable in the `Config` object of `ysyxSoC/src/Top.scala` to `true`. After modification, regenerate `ysySoCFull.v` and try running some test programs.

Enable burst transfer for icache

Modify the implementation of icache so that it uses burst transfer to access data blocks in SDRAM.

After the implementation is correct, record the waveform of the burst transfer and compare it with the waveform when burst transfer is not used. You should be able to observe that using burst transfer does indeed improve efficiency.

Calibrate the memory access delay of the AXI interface SDRAM

Although the burst transfer mode is currently working, the corresponding memory access delay has not been calibrated, so the performance data is inaccurate. Similar to the previously implemented APB delay module, we need an AXI delay module to calibrate the corresponding memory access delay.

Since the AXI protocol is more complex than the APB protocol, the implementation of the AXI delay module requires additional consideration of the following issues:

- AXI read and write channels are independent, so in principle, separate counters should be designed to control the delay for read and write transactions. However, the current NPC is multi-cycle and does not send read and write requests simultaneously, so you can currently design a unified counter to control both. However, when implementing the pipeline later, you will still need to use two independent counters.
- AXI has complete handshake signals, and waiting for the handshake also involves the device's state, so this period should also be included in the calibration range, i.e., the moment when the valid signal is active should be considered the start of the transaction.
- AXI supports burst transfers, so the transfer mode differs from APB.
- Taking a read transaction as an example, burst transfers may involve multiple data receptions, each of which requires separate synchronization. Assume an AXI burst read transaction starts at time `t0`. The device receives data at times `t1`, `t2`, and the AXI delay module sends data upstream at `t1'` and `t2'`. Then, the equations `(t1 - t0) * r = t1' - t0` and `(t2 - t0) * r = t2' - t0` should hold.
- Burst transmission of write transactions involves multiple data transmissions. Since the device needs at least one cycle to receive data, the CPU considers that `r` cycles have already elapsed, so the data transmission time also needs to be calibrated. However, since we have not yet implemented dcache, LSU does not initiate burst write transactions, so burst write transaction calibration can be temporarily omitted. However, calibration is still required for single write transactions.

Implement AXI delay module

Based on the previous description, implement the AXI delay module in ysyxSoC. Specifically, if you choose Verilog, you need to implement the corresponding code in `ysyxSoC/perip/amba/axi4_delayer.v`; If you choose Chisel, you need to implement the corresponding code in the `AXI4DelayerChisel` module of `ysyxSoC/src/amba/AXI4Delayer.scala`, and modify `Module(new axi4_delayer)` in `ysyxSoC/src/amba/AXI4Delayer.scala` to instantiate the `AXI4DelayerChisel` module.

To simplify the implementation, you can currently assume that the number of burst transmission cycles will not exceed 8. After implementation, try different values of `r` and observe whether the above equation holds true in the waveform.

Evaluate the performance of the burst transmission method

After calibrating the memory access delay of the burst transmission method, run the microbench train scale test and compare the results with the previous records.

Quickly evaluate the cost of missing

Based on the previous discussion, the current missing cost of the ICache is only related to the block size and the bus transmission mode, and is independent of other cache parameters. Therefore, we can pre-evaluate the missing cost for various combinations of block sizes and bus transmission modes, and directly substitute this missing cost into the formula to calculate the TMT, thereby estimating the expected benefits of different cache parameter combinations. However, based on the above analysis, burst transmission mode will definitely perform better than independent transmission mode. Therefore, in practice, we only need to evaluate the miss cost of various block sizes in burst transmission mode in advance.

There are two methods for evaluating the miss cost in advance:

1. Modeling. Based on the workflow of the SDRAM controller state machine, derive a formula for calculating the SDRAM access time. Substitute the block size to calculate the corresponding miss cost. This method is relatively straightforward, but the accuracy of the model is a challenge. For example, the row buffer and refresh operations of SDRAM also affect SDRAM access time, but it is difficult to quantify the overhead these factors introduce in a single SDRAM access.

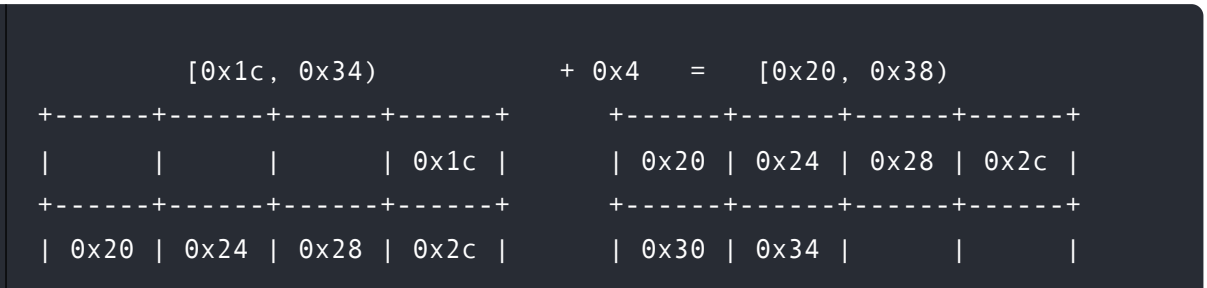
2. Statistical analysis. By using appropriate performance counters, calculate the TMT when accessing SDRAM with an ICache miss, and then calculate the average miss cost. As a statistical method, it can consider factors that are difficult to model, such as SDRAM row buffers and refresh operations, by averaging the results of a sample. However, the row buffer is essentially a cache, and its performance is also affected by program locality. Therefore, the test program must be representative: Running the representative train-scale test of microbench is the best way to ensure representativeness, although it takes more time. Running the train-scale test of the microbenchmark directly provides the best representativeness, though it requires more time, it only needs to be run once to calculate the average miss cost; The behavior of the test-scale test is not entirely the same as that of the train-scale test, but it still has some representativeness, allowing for a quick calculation of the average missing cost; However, the representativeness of the hello program is relatively weak, and using it to estimate the average missing cost may result in significant errors.

In actual projects, considering the complexity of the project, modeling is rarely used. Therefore, we also recommend using the statistical method to pre-assess the missing cost.

> **Quickly assess the missing cost**
>
> Implement a quick assessment of the miss cost based on the above content. You can already calculate the number of misses using cachesim, and you will use these miss costs to estimate the expected benefits of different cache parameter combinations.

Program memory layout

The memory layout of a program can also significantly affect cache performance. Take icache as an example. When the icache block size exceeds 4 bytes, some hot loops in the program may not be aligned with the cache block boundaries, causing the instructions in the hot loops to occupy additional cache blocks. For example, suppose a hot loop is located at address `[0x1c, 0x34)`, and the block size of an icache is 16 bytes. To read all the instructions in this hot loop into the icache, three cache blocks are required.



```
+------+------+------+------+      +------+------+------+------+
| 0x30 |      |      |      |
+------+------+------+------+
```

However, if we fill in some blank bytes before the program code, we can change the position of the hotspot loop, so that the instructions in the hotspot loop occupy fewer cache blocks. For example, in the above example, we only need to fill in 4 bytes of blank content before the program code, to change the position of the hotspot loop to `[0x20, 0x38)`. At this point, the hotspot loop instructions only occupy 2 cache blocks, and the 1 cache block saved can be used to store other instructions, thereby improving the overall performance of the program.

> **Optimize the memory layout of the program**
>
> Try filling the program code with a few blank bytes using the method described above. Specifically, you can achieve this by modifying the code or modifying the link script. After implementation, try evaluating whether the filled blank bytes optimize the program's performance.

In fact, the cache capacity in the above example is very small, so the 1 cache block saved accounts for a high proportion of the entire cache. However, modern processors have relatively large cache capacities, so saving one cache block may not significantly improve program performance. Nevertheless, we would like to point out that program optimization is an important direction for utilizing the principle of locality. Some programs can even achieve several times the performance after optimization.

In enterprises, for key applications in target scenarios, engineering teams usually use various methods to improve their performance. If an enterprise has the capability to design its own processors, in addition to improving processor performance, it will also customize the compiler based on the processor parameters. Compared to executable files compiled using public versions of compilers (such as gcc from the open-source community), executable files compiled using customized compilers can run faster on the target processor. Specifically, **SPEC CPU defines two metrics, `base` and `peak`**, where the `peak` metric allows different sub-items to be compiled with different compilation optimization options, enabling the entire benchmark to achieve better performance on the target platform than the `base` metric. If you want to achieve a higher score in the `peak` metric, software-level optimization is indispensable.

Design Space Exploration (2)

We have introduced many cache parameters above, including the memory layout of the program, which all affect the performance of the program running on the processor. Now we can comprehensively consider these parameters and select a set of parameter combinations with better performance. Of course, design space exploration also needs to satisfy area size constraints.

! Area size constraints

Your NPC needs to be implemented on the nangate45 process provided by the `yosys-sta` project by default, with a total area not exceeding 25,000 um^2 (the default unit for area data reported by the synthesis tool is um^2), which also serves as the area constraint for the B-stage tape-out. Considering that subsequent tasks will require the implementation of pipelining, we recommend that the total area of the NPC does not exceed 23,000 um^2 at this stage.

This area is not particularly large, on the one hand, adopting this area limit helps highlight the contribution of other parameters in the design space exploration, otherwise, you could simply increase the cache capacity to achieve good performance, and the impact of other parameters on performance improvements would be difficult to discern; on the other hand, the project team expects many students to base their work on the B-stage tape-out, a smaller area helps the project team save tape-out costs.

Currently, you do not need to strictly adhere to the above area constraints. If the excess is less than 5%, you can choose to perform unified optimization after completing the pipeline; however, if the current total area significantly exceeds the above constraints, you may need to make significant adjustments to your design, and we recommend that you immediately initiate area-related optimization efforts.

Regarding wafer fabrication costs, we can make some simple but not very rigorous estimates. Assuming a foundry offers nangate45 process technology with a block size of `2mm x 3mm` at a price of 500,000 CNY, the cost per um^2 is `500,000 / (2 * 3 * 1,000,000) = 0.0834` CNY. Typically, the interconnecting lines between standard cells also occupy a certain amount of area, and to prevent excessive congestion, additional space is left between standard cells. Therefore, the final chip

area is usually larger than the synthesis area. Based on experience, the synthesis area is generally 70% of the final chip area. Using the above estimation method, a design with a synthesis area of 25,000 um^2 would require a final cost of `25,000 / 0.7 * 0.0834 = 2,978` CNY.

The key takeaway from this estimation is that adding functional features to a CPU is not free. This is very different from FPGA-based designs: In some competitions targeting FPGA platforms, participants typically strive to convert as much FPGA resources into CPU performance as possible, without incurring any economic costs in the process.

However, real-world chip production is not like this. You can view the goal of Phase B as designing a low-cost embedded processor(rv32e is a basic instruction set for embedded scenarios): Assume you are an architect at an embedded CPU manufacturer. You need to find ways to improve processor performance within a limited area budget. If the area exceeds expectations, the chip cost will increase, and its competitiveness in the market will decrease. Under these conditions, you need to estimate the cost-effectiveness of adding a feature: Assuming that a certain feature can improve performance by 10%, is it worth paying an additional 500 CNY for it?

Explore the design space of icache

Based on the above introduction, explore the design space of icache and determine a design plan that achieves good performance under the given constraints. After determining the solution, implement it in RTL and evaluate its performance in ysyxSoC.

Some ideas for optimizing area

If your estimated area greatly exceeds the above requirements, you will most likely need to optimize your design. There are no tricks to optimizing area, but overall, you can consider the following directions:

1. Optimize logic overhead: Consider which logic functions are redundant and can be merged with existing logic
2. Optimize memory overhead: Consider which memory units are redundant

3. Conversion between logic overhead and memory overhead: Sometimes, instead of storing a signal, it is better to recalculate it, though this may affect the critical path and requires case-by-case analysis

For most people, meeting the above area constraints is unlikely to be achieved with a quick write, but it is not impossible. yzh's reference design achieves an area of 22,730 um^2 after adding an ICache, while maintaining a frequency of 1,081 MHz, and the `Total time` displayed in the microbenchmark is 4.49 seconds. We set the above area constraints for two reasons: If you are a beginner, you need an opportunity to get started with this kind of work, and through trial and error, you will develop an understanding of the relationship between each line of RTL code and its area overhead.

On the other hand, it is also to remind everyone of the goal of architectural design: Performance and area are mutually constrained. If you find your design difficult to optimize, the simplest method is to reduce the cache capacity, but you will have to sacrifice performance; If you want to balance area and performance, you need to minimize unnecessary area overhead, and then plan how to use this area to better improve performance.

! Do not focus too much on synthesis options

Some students attempt to improve synthesis quality by exploring various options in the synthesizer. While we encourage everyone to understand the details of synthesis, it is important to note that improving synthesis quality by modifying the code is different from improving synthesis quality by tweaking parameters. In fact, the latter is generally only done when pursuing extreme optimization goals, but the former must be done well first. More fundamentally, the former belongs to the training of architectural design skills, while the latter does not.

Evaluate the cost-effectiveness of dcache

You have estimated the performance gain of dcache under ideal conditions in the previous section. Here, we will continue to estimate the cost-effectiveness of this dcache. Assuming that the area of this dcache is the same as that of the icache, what is its cost in CNY?

Although you have not yet designed the dcache, since it needs to support write operations, its design must be more complex than that of the icache, and therefore it should occupy a larger area than an icache of the same capacity. Therefore, the cost-effectiveness of the dcache estimated under these conditions is highly optimistic. If we consider the actual performance gains and physical area of the dcache, the cost-effectiveness of designing a dcache will be even lower.

Another direction to consider is: if the area of the dcache is used to expand the capacity of the icache, how much performance improvement can be achieved?

📢 Real computer architecture design

Although the above icache design space exploration task is significantly simplified compared to real processor design, it is still the first time most students have encountered real processor architecture design. More importantly, this is likely the first time most students have experienced the entire design process of a module, from requirement analysis, structural design, logical design, to functional verification, performance verification, performance optimization, finally to area evaluation and timing analysis at the circuit level. Among these, logic design is what is commonly referred to as RTL coding.

This task once again demonstrates that computer architecture design is not equivalent to RTL coding. The work of computer architecture design is to find a set of design parameters with good performance within the design space that satisfies the constraints. However, the design space is typically very large, and thoroughly evaluating the performance of a set of design parameters can be time-consuming. Therefore, for computer architecture design, the ability to quickly assess the performance of different design parameters is a critical issue.

Therefore, simulators are an essential tool for computer architecture design. With simulators, we do not need to simulate circuit-level behavior (no Verilator runs, only CacheSim runs), we only need to simulate necessary modules (no cache data simulation, only cache metadata simulation), and we do not need processor-driven simulation (no full program runs, only playback of corresponding ITrace). It is these differences that make simulators significantly more efficient than RTL simulation, enabling rapid evaluation of the expected performance of different design parameters and helping us quickly eliminate obviously unsuitable design parameters.

According to the Xiangshan team's experience, running a program on Verilator takes one week, but running the same program on a full-system simulatorgem5, the same program can be run in just 2 hours. This means that the time required to evaluate a set of design parameters in an RTL simulation environment can be used to explore the effects of 84 different design parameter combinations using the simulator.

For architectural research, simulators are also a common platform. The ISCA conference has held multiple competitions based on the ChampSim simulator, including the cache replacement algorithm competition and the data prefetch algorithm competition. Researchers evaluate the performance of various algorithms in the simulator to quickly adjust the overall implementation and fine-tune the parameters of the algorithms. Although a qualified algorithm still requires implementation and verification at the RTL level, exploring various algorithms at the RTL level from the outset is highly inefficient.

Therefore, when you truly understand that `computer architecture design ≠ RTL coding`, you have truly begun your journey into the field of computer architecture design.

Cache Coherence

When a store instruction modifies the content of a data block, according to the program's semantics, subsequent instructions should read the new data from the corresponding address; otherwise, the program execution will fail. However, due to the cache mechanism, multiple copies of the data may exist in memory. Ensuring that the new data is read from each copy is known as the cache coherence problem.

In computer systems, from the cache in a processor to distributed systems and the internet, whenever there are data copies, there will be consistency issues between them. After adding icache to NPC, we can reproduce a consistency issue using the following `smc.c`:

```c
// smc.c
int main() {
asm volatile("li a0, 0;"
"li a1, UART_TX;" // change UART_TX to the correct address
"li t1, 0x41;" // 0x41 = 'A'
"la a2, again;"
"li t2, 0x00008067;" // 0x00008067 = ret
"again:"
```

```
  "sb t1, (a1);"
  "sw t2, (a2);"
  "j again;"
  );
  return 0;
  }
```

The above program first initializes some registers, then writes the character 'A' to the serial port at the label 'again', then rewrites the instruction at 'again' to 'ret', and finally jumps back to 'again' to re-execute. According to the program's semantics, the program should output the character `A` and then return to `main()` via the modified `ret` instruction. This type of code that modifies itself during execution is called "self-modifying code" (Self Modified Code).

Self-modifying code in the history of computer development

In the past, when memory address space was extremely limited, self-modifying code was often used to improve memory utilization, enabling programs to perform more functions within limited memory. For example, the **FC NES console from the 1980s had only 64KB of address space**, with 32KB occupied by the ROM on the cartridge. Some cartridges also had 8KB of RAM, but if the cartridge did not have RAM, the program could only use the 2KB of RAM integrated into the CPU. To develop exciting games with such limited resources, developers employed numerous cutting-edge techniques, and self-modifying code was one of them.

With the development of memory technology, memory capacity is no longer as limited as it was in the past, and since self-modifying code is difficult to read and maintain, it is now rarely seen in modern programs.

Reproduce cache consistency issues

Compile the above program for the AM and run it on the NPC. What issues did you encounter? Try to analyze the issues and verify your findings using waveforms.

If you did not encounter any issues, try increasing the ICache capacity.

To solve the above problem, a direct solution is to ensure that all copies in the system are always consistent. For example, each time a store instruction is executed, immediately

check whether there are other copies in the system. If there are, update them or invalidate them to ensure that subsequent operations, regardless of where they are accessed from, can directly read the new data (using the update method) or read the new data from the next storage level due to its absence (using the invalid method). The x86 instruction set adopts this solution. However, this clearly increases the complexity of CPU design. In particular, in some high-performance processors, when a store instruction is executed, other components also access various caches in the system at the same time. It is very challenging to prevent other components from accessing outdated data before the store instruction completes the update or invalidation of all copies.

Another solution is more lenient. allowing copies in the system to be inconsistent at certain times, but before the program accesses this data block, a special instruction must be executed to instruct the hardware to handle outdated copies. In this way, the program's execution can still access the correct data, and the results remain consistent with the program's semantics. The RISC-V instruction set adopts this approach. RISC-V includes a `fence.i` instruction, whose semantics ensure that all subsequent fetch operations can see the data modified by store instructions that occurred before it. Here, the `fence.i` instruction acts as a barrier, preventing subsequent instruction fetches from crossing the barrier to read the old data modified by the store instruction. Additionally, the RISC-V manual states:

```
RISC-V does not guarantee that stores to instruction memory will be
made
visible to instruction fetches on a RISC-V hart until that hart
executes
a FENCE.I instruction.
```

In other words, RISC-V allows the copy in the icache to be inconsistent with memory at certain times, which is consistent with the discussion above. For more information about `fence.i`, please refer to the manual.

RISC-V only defines the semantics of `fence.i` at the instruction set level, but there are multiple different schemes for implementing the functionality of `fence.i` at the microarchitecture level:

Scheme	When executing a store instruction	When executing `fence.i`	When accessing the icache
(1)	Update the corresponding block in the icache	nop	Hit

Scheme	When executing a store instruction	When executing `fence.i`	When accessing the icache
(2)	Invalidate the corresponding block in the icache	nop	Miss, access memory
(3)	-	Flush the entire icache	Missing, access memory

In fact, implementation schemes (1) and (2) are the “keep all copies in the system consistent at all times” scheme mentioned above, which is a special case of “allow copies in the system to be inconsistent at certain times”: Since all copies are already consistent when executing the store instruction, `fence.i` can be implemented as `nop`. For scheme (3), the copies in the icache are not processed when the store instruction is executed, so these copies remain in an inconsistent state. Therefore, when executing `fence.i`, the icache must be flushed to achieve its barrier effect, ensuring that subsequent accesses to the icache are missing, thereby accessing the new data in memory. However, regardless of the implementation scheme chosen, the program must include the `fence.i` instruction to meet the requirements of the RISC-V manual. Otherwise, the program will not run correctly on processors implementing scheme (3).

In fact, the difference between these implementation schemes lies solely in whether the consistency of copies is handled at the hardware level or the software level: If the instruction set specification requires the processor to handle copy consistency at the hardware level, this issue is transparent to the software, and programmers do not need to consider where to add instructions like `fence.i` in the program, but the trade-off is more complex hardware design; If the instruction set specification requires the processor to handle copy consistency issues at the software level, the hardware design is simpler, but the cost is increased programmer burden. Therefore, the essence of this issue is a trade-off between hardware design complexity and programmer burden.

Implement the `fence.i` instruction

Based on your understanding of the `fence.i` instruction, choose a reasonable approach to implement it in NPC. After implementation, add the `fence.i` instruction to the appropriate location in the aforementioned `smc.c` file and rerun NPC. If your implementation is correct, you will see the program output the character `A` and exit successfully.

> Hint: You may encounter compilation errors related to `fence.i`. Try to resolve the issues based on the error messages.

In fact, cache consistency issues manifest in more ways on real computers. As processors become more complex, we will discuss additional cache consistency issues in future discussions.