Revisiting Trace Cache

Hrishikesh Belatikar *belatikar@wisc.edu*

Anna Iwanski *ariwanski@wisc.edu*

Hailey Johnson *hljohnson22@wisc.edu*

Jagdish Mohapatra *nmohapatra@wisc.edu*

Abstract

A trace cache stores instructions dynamically, based on the order of their execution, rather than the order in which they appear in memory and are compiled [1]. Trace Caches were found to be especially effective for wide issue machines as they improve instruction fetch bandwidth [2]. They were commercially implemented two decades ago by Intel in their Pentium 4 processor line. Unfortunately, trace caches were discontinued majorly due to their high requirement of on-chip area. But recently, with smaller ultra-low power node technologies like sec5lpe/4lpe (Samsung's 5nm/4nm) and TSMC 5FF, designers have more flexibility in terms of area and trace caches may be a viable solution in increasing performance. This report discusses the re-implementation of a trace cache and how it was modeled within the existing TimingSimpleCPU in the gem5 simulator [3]. We used GAPBS and SPEC06 benchmark suites for evaluating the performance improvement when a trace cache is implemented.

1 Introduction

Instruction fetch bandwidth is becoming a performance bottleneck which is a concern. Instruction fetch performance depends on several factors like instruction issue rate, fetch unit latency, branch throughput, etc. Instruction cache hit rate and branch prediction accuracy have long been recognized as important problems in fetch performance. One potential solution to this concern is the reintroduction of the trace cache.

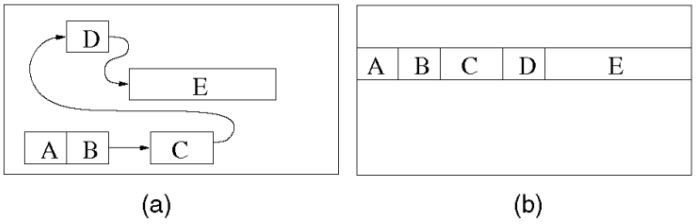

Figure 1: Storing a noncontiguous sequence of instructions. (a) Instruction cache. (b) Trace cache. (Adapted from [2])

The trace cache is a special small instruction cache which captures dynamic instruction sequences or ‘traces’ opposed to a traditional instruction cache where instructions are kept in their compiled order, shown in Fig. 1. There has been some work attempting to store instructions within the instruction cache in an order closer to their execution order to decrease fetching latencies, however it is not possible to fully store a program in execution order before it has executed. This is where the trace cache can assist in decreasing fetching latencies. It works alongside an instruction cache in the fetch stage of the pipeline to potentially allow for faster fetching of repeated branch instructions with their basic blocks. The trace cache capitalizes on the knowledge that most programs will repeat the same block of code many times, therefore the trace will have many misses at first, with the assumption that those misses will be hits in the future.

A trace is a sequence of at most ’n’ instructions or at most ’m’ basic blocks starting at any point in the dynamic instruction stream as shown below in Fig. 2. The limit ’n’ is the trace cache line size, and ’m’ is the branch predictor throughput. A trace is fully specified by a starting address and a sequence of up to ’m-1’ branch outcomes which describe the path followed. The first time a trace is encountered, it is allocated a line in the trace cache. The line is filled as instructions are fetched from the instruction cache.

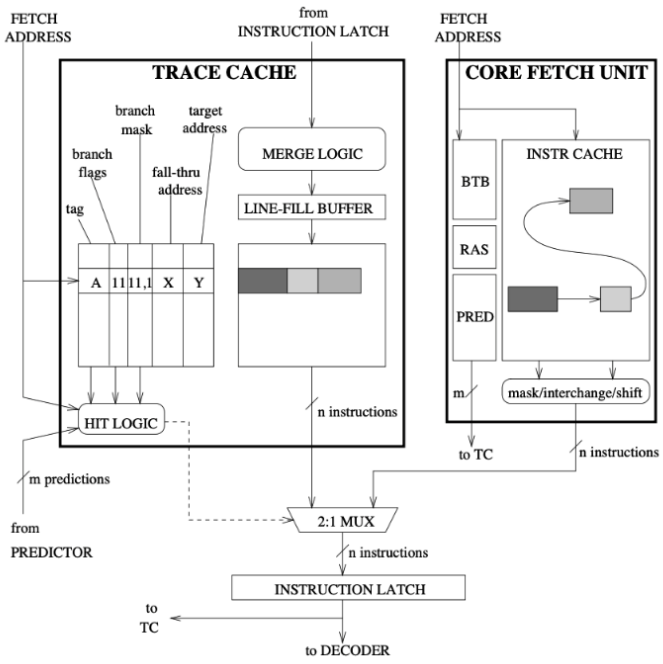

Figure 2: Trace cache implementation in detail (Adapted from [1])

If a branch is encountered during the course of program execution and the trace cache unit is not currently in use, the trace cache will be searched for a hit using the starting address of the given branch and branch prediction outcomes for the next m-1 branches. On a hit, the full trace is fed directly to the decoder bypassing the instruction cache. Otherwise, it is considered a miss and fetching proceeds normally from the instruction cache while the trace cache unit begins constructing a new trace to store for potential future use.

2 Related Work

The micro-architecture of the trace cache is discussed in [2], used the SimpleScalar simulator to model the trace cache and its behavior. Their model included a trace predictor and outstanding trace buffers used to construct new traces that are not in the trace cache and track branch outcomes as they become available from the execution engine. They used the six integer SPEC92 benchmarks and six benchmarks from the Instruction Benchmark Suite (IBS) to evaluate the performance of the various fetch mechanisms in their implementation. The authors of [4] have proposed methods to improve trace cache efficiency. They found that most traces are rarely used in the trace cache again before being replaced and most of the instructions delivered for execution originate from the fewer traces that are heavily used. They suggested splitting the trace cache into two components: the filter trace cache and the main trace cache in [4]. Traces are first inserted into the filter trace cache that is used to filter out the infrequently used

traces; traces that prove useful are later moved into the main trace cache itself. As a result, their approach decreases the number of traces built, thus reducing power consumption while improving overall performance.

In our project, we introduced the trace cache into the gem5 simulator environment and ran our changes on some latest SPEC benchmarks to evaluate the trace caches performance with the built-in CPUs.

3 Proposed Approach

Our approach is to model a simple trace cache in the gem5 simulator and perform benchmark testing to determine the viability of adding a trace cache to a new hardware system. The Trace Cache (T$) works in parallel with the Instruction Cache (I$), i.e., the addresses will be fetched by the I$ and fed into the T$ where logic will determine what steps to take. This will allow us to measure the number of T$ hits and misses for every indexing opportunity. The T$ will provide no output back to the CPU and does not affect the functionality of the system, it only provides data for further analysis by the user. This allows the development to focus on the indexing logic rather than implementation or integration. It is only designed to work with the TimingSimpleCPU model provided by gem5, which is an in-order pipelined processor model.

We are using the SimpleScalar based trace cache implementation [5] to jump start the modelling process of the trace cache in gem5. The system setup such as L1 cache width, size, trace length and associativity are adapted from [6]. The above parameters are used to perform a subset of the SPEC CPU benchmarks which target the branch predictor. We are also using the gapbs software to benchmark the system due to its ease of use. To gain meaningful insights we are observing the cache hits and misses for our implemented trace cache. The statistics provided by the gem5 cache would work as a template in logging the results for our cache. The statistics would be generated by running the SPEC benchmark suite applications on the TimingSimpleCPU in [4] the gem5 simulator.

4 Methodology

Our methodology to generate and analyze the trace cache using gem5 was a three-step approach: configure the gem5 environment to work with SPEC06 benchmarks, generate the logic of trace cache functionality, and integrate the trace cache logic as a model in gem5.

4.1 gem5 Environment Setup for SPEC06 Benchmarking

Our benchmarking process uses the following script/config files:

- **spec06_config.py:**

A modified version of gem5's "se.py" which takes in new arguments such as which SPEC benchmark to run, where to write the output and the errors.

- **run_gem5_x86_spec06_benchmark.sh:**

A bash script which invokes gem5 and runs a specific SPEC06 benchmark and stores the output in its respective directory. It takes the benchmark name and its respective run directory as the arguments.

4.2 Developing a Gem5 Model for Trace Cache

To implement the trace cache functionality as per our scope, we designed the process flow as shown in Figure 3The logical flow described in Figure 3 models a trace cache which allows one branch instruction per trace. The scope is reduced from the initial three branches per trace line to a single branch due to complexity of the logic and time constraint. Furthermore, this trace cache model stores speculative traces and utilizes a random eviction policy.

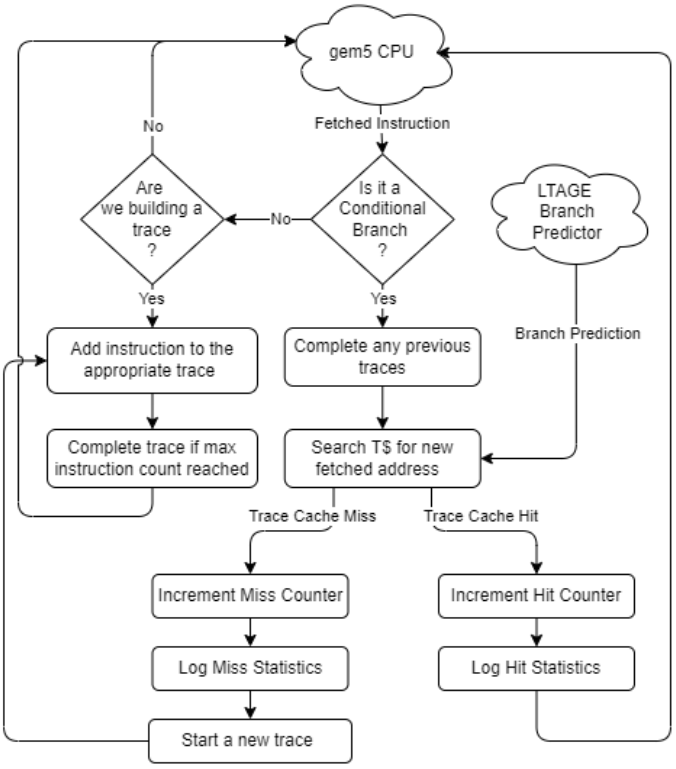

Figure 3: Trace Cache Logic Flow Chart

Our trace cache only models the number of times there is a cache hit or miss and does not affect the functionality of the system. Thus, instructions are not stored in our cache and only the control and indexing logic is simulated. On every instruction fetch, the trace cache is accessed with the PC of the most recent instruction fetch,

information on whether the instruction fetched is a branch instruction, and the branch prediction of the fetched instruction. If the most recently fetched instruction is not a branch instruction, then the branch prediction value is zero.

If the instruction is not a branch and a trace is currently being built, then the instruction is added to the trace being built. If the instruction is not a branch and there is no trace currently being built, then no action is taken. If the instruction is a branch, the trace cache is indexed using a hashing logic of the program counter (PC) address and the branch predictor output. Thus, a branch can have both the taken and not taken traces present in the trace cache. Our model only starts generating traces if the system encounters a branch instruction.

The trace cache access is a hit if the valid bit in the indexing address is a one, otherwise it is a miss. If the trace cache misses, the miss counter is incremented, the current trace being built is completed and the building of a new trace begins. This is because the modeled trace cache only includes one branch instruction per trace. A trace continues to be built for each newly fetched instruction if the number of instructions within the trace has not reached the maximum number of instructions allowed in a trace and as long as a branch instruction has not been fetched. This new trace would be available to use the next time the same branch instruction is fetched. If a trace cache hit occurs, the hit counter is incremented.

This logic flow can be utilized on trace cache models of varying organization and capacity; thus, the logic flow is integrated with a parameterizable set-associative cache structure. By modifying the number of sets and the associativity of a set-associative cache, all basic cache configurations can be explored, including direct-mapped caches, set-associative caches, and fully associative caches. Exposing these configurations to the user allows simple design space exploration without significant additional effort.

5 Implementation

The logical flow of the trace cache model described in the Methodology Section is implemented utilizing C++ classes and is integrated into the gem5 simulator.

5.1 Trace Cache Model Implementation

The trace cache model is implemented via two C++ classes. The tcLine class models one line within the trace cache by storing relevant flags and the number of instructions within the trace of the given trace cache line. The traceCache class models a trace cache via an array of tcLine objects. Figure 4 displays both C++ classes in a UML diagram.

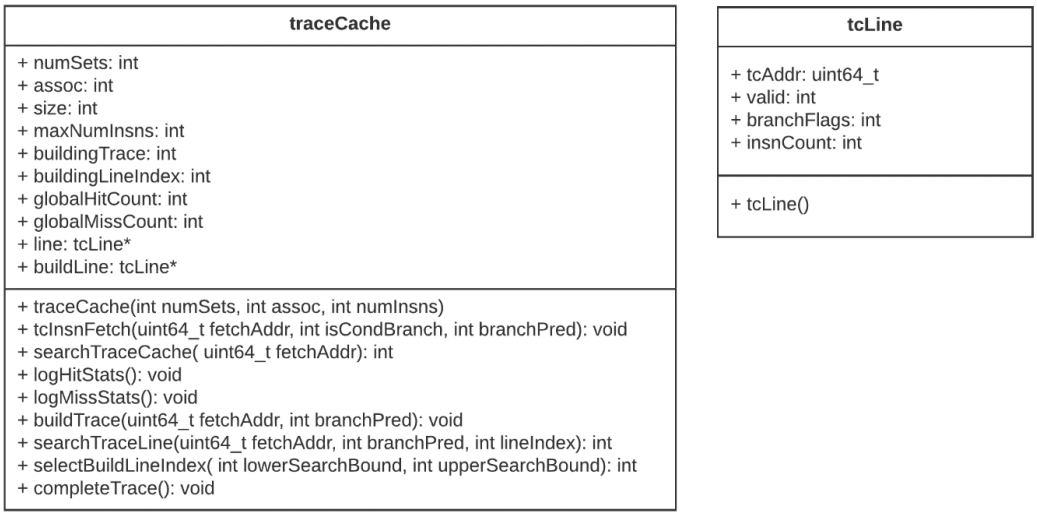

Figure 4 : UML Diagram of traceCache C++ class and tcLine C++ class

5.1.1 Trace Cache Line (tcLine)

Within a trace cache line, the following fields are stored:

- PC Address (tagAddr): *Unsigned 64-bit Integer*. A trace cache line is indexed partially via the PC address of the most recent instruction fetch. When the population of a new trace cache line has begun, this PC address is saved. It is used for the search of the trace cache during future instruction fetch operations.
- Valid (valid): *Integer*. A valid bit for a given trace cache line modeled using an integer. This value is used during trace cache search.
- Branch Flags (branchFlags): *Integer*. One bit is allocated for each branch instruction within a trace. In this trace cache model, there is only one branch instruction within a trace, so branchFlags is a one-bit value modeled as an integer. branchFlags is populated with the branch prediction of the tagAddr when the construction of a new trace begins.
- Instruction Count (insnCount): *Integer*. Stores the number of instructions within a given trace cache line.

The fields tagAddr, valid, and branchFlags directly map to fields stored within a trace cache line in previous trace cache implementations. The insnCount field is not explicitly mentioned in trace cache literature; however, it is a necessary field for tracking the number of meaningful instructions within a trace cache line. One noteworthy point is that the trace cache line class does not store any instructions, as would be necessary in a useful trace cache line implementation. In order to generate simple cache performance metrics such as cache hit rate and miss rate, a detailed

representation of the instructions within a trace is not necessary. By implementing only, the trace cache logic and saving the trace cache line flags, the trace cache model is simple and lightweight.

5.1.2 Trace Cache (traceCache)

The traceCache class contains an array of trace cache lines, some supporting fields, and functions to implement the trace cache logic. It is implemented as a parametrizable, set-associative cache. Within the constructor of the class, one can specify the number of sets within the cache, the associativity of the cache, and the maximum number of instructions that can be held within a trace. The constructor for the traceCache class is provided within Figure 4. Figure 5 displays visually how the parameters set within the trace cache constructor influence the size and organization of the cache. In Figure 5, each wide rectangle represents a trace cache line, and each square represents an instruction held in a given trace. This example cache has three sets, an associativity of four, and allows a maximum of twelve instructions per trace.

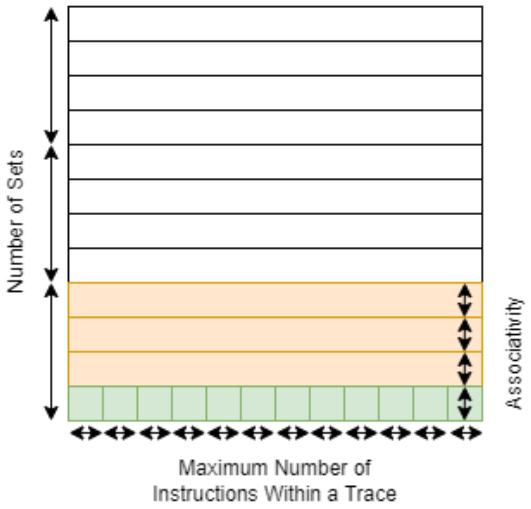

Figure 5 : Parameterizable Trace Cache

The traceCache object has many fields and functions which are utilized throughout modeling; however, not all functions and fields within this class need to be mentioned to understand the basics of the implementation. Within the traceCache class the following notable fields are stored, and notable functions are exposed to the user.

Notable Fields Within the traceCache Class:

- Global Hit Count (globalHitCount): *Integer*. A counter which is incremented whenever a trace cache hit occurs.

- Global Miss Count (globalMissCount): *Integer*. A counter which is incremented whenever a trace miss occurs.
- Array of Trace Cache Lines (line): *tcLine**. A reference to the array of tcLine objects. This array models the internals of the trace cache and is the backbone of the trace cache model.
- Build Trace Cache Line (buildLine): *tcLine**. A tcLine object which is used to build a new trace. The trace being built is built outside of the array of tcLine objects. Once the trace being built is completed, its fields are copied into the array of trace cache lines.
- Number of Sets (numSets): *Integer*. A field set via the constructor of the traceCache class. Denotes the number of sets within the set-associative trace cache model and is used to create the array of tcLine objects.
- Associativity (assoc): *Integer*. A field set via the constructor of the traceCache class. Denotes the associativity of the set-associative trace cache models and is used to create the array of tcLine objects.
- Size (size): *Integer*. The size is calculated as numSets*assoc. It is the number of tcLine objects within the trace cache.
- Max Number of Instructions (maxNumInsns): *Integer*. A field set via the constructor of the traceCache class. Denotes the maximum number of instructions which can be held within a trace.

Notable Functions Within the traceCache Class:

- Constructor (traceCache()): The constructor for the traceCache object creates a parameterizable interface for creating traceCache objects of various cache configurations.
- Trace Cache Access Function (tcInsnFetch()): This function performs the logic outlined in the Methodology section. It is called on every instruction fetch.
- Trace Cache Search (searchTraceCache()): This function is called when the instruction fetched is a branch instruction. Utilizing other helper functions, this function returns a 1 if a trace cache hit occurs and a 0 if a trace cache miss occurs.
- Eviction Policy Implementation (searchTraceLine()): This function is called when a trace cache line needs to be evicted. The replacement policy first looks for any invalid traces. If an invalid trace is found, it is replaced with a new trace. If no invalid traces are found, then a random trace within the correct set is evicted.

5.2 gem5 Integration

An ideal implementation would not require modifications to a specific CPU; however, due to challenges highlighted in the Challenges Section, the implementation is closely tied to a given CPU. The trace cache is integrated into and simulated using the TimingSimpleCPU. The TimingSimpleCPU is an in-order, unpipelined CPU within the SimpleCPU classification [9]. There are two CPU versions that are derived from BaseSimpleCPU: TimingSimpleCPU, and AtomicSimpleCPU [9]. This means changes had to be made to the BaseSimpleCPU classes as well as the TimingSimpleCPU classes, while not changing the implementation for the AtomicSimpleCPU, to integrate the trace cache implementation for testing.

5.2.1 BaseSimpleCPU Changes

A traceCache object for every configuration of interest was instantiated within the constructor of the BaseSimpleCPU. Pointers to these traceCache objects are public fields of the BaseSimpleCPU, making them accessible to the TimingSimpleCPU. A simple code snippet illustrating these changes is included in Figure 6.

```
// within BaseSimpleCPU constructor in base.cc
    tc_dm = new traceCache(64,1,16);
    tc_fa = new traceCache(1,64,16);
    tc_sa1 = new traceCache(32,2,16);
    tc_sa2 = new traceCache(16,4,16);
```

```
// within BaseSimpleCPU constructor in base.hh
public:
    traceCache *tc_dm;
    traceCache *tc_fa;
    traceCache *tc_sa1;
    traceCache *tc_sa2;
```

Figure 6: Left Image: traceCache object instantiation. Right Image: public pointers to all traceCache objects

In this example, four trace cache configurations are studied. All four cache configurations can hold 64 traces and 16 instructions per trace. A direct mapped cache is modeled as tc_dm, a fully-associative cache is modeled as tc_fa, and two set-associative caches modeled as tc_sa1 and tc_sa2 where tc_sa1 has an associativity of 2 and tc_sa2 has an associativity of 4. These traceCache objects will be used in code snippets and diagrams moving forward to provide a simple illustration of the implementation.

Furthermore, the preExecute() function within the BaseSimpleCPU is modified. The original definition of the preExecute() function does not take any inputs, but it is modified to take a PacketPtr object as an input to pass to the trace cache. The PacketPtr object is used to access the given instructions PC address for storage in the trace cache. To keep the functionality of AtomicSimpleCPU, we overloaded the preExecute() function by creating one that takes in a PacketPtr object and uses it and one that goes unchanged. Within the preExecute() function, the PC Address, information on the whether the instruction is a branch instruction, and the branch prediction of the

instruction, are used to call the tcInsnFetch() function if available. Each traceCache object will be called to further deduce next steps of the instruction if a new instruction has been fetched. A simple code snippet illustrating these modifications to the preExecute() function is provided in Figure 7.

```
if(curStaticInst && pkt){
       tc_dm->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
       tc_fa->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
       tc_sa1->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
       tc_sa2->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
}
```

Figure 7: tcInsnFetch() is called for all traceCache objects within the preExecute() function in TimingSimpleCPU.

5.2.2 TimingSimpleCPU Changes

Because the TimingSimpleCPU is a child class derived from the BaseSimpleCPU, minimal changes were made to the TimingSimpleCPU. The only change required was a modification to the preExecute() function call within the completeIFetch() function call within the TimingSimpleCPU. This function call was modified to pass a pkt as an input. Figure 9 illustrates how the changes to the preExecute() function appear within the larger execution flow of the TimingSimpleCPU. The TimingSimpleCPU has two stages: Fetch and Execute. The completeIFetch() function is called within the fetch stage, signaling a transition to the execute stage. It is in the completion of the fetch stage prior to the beginning of the execution stage that the traceCache objects are accessed.

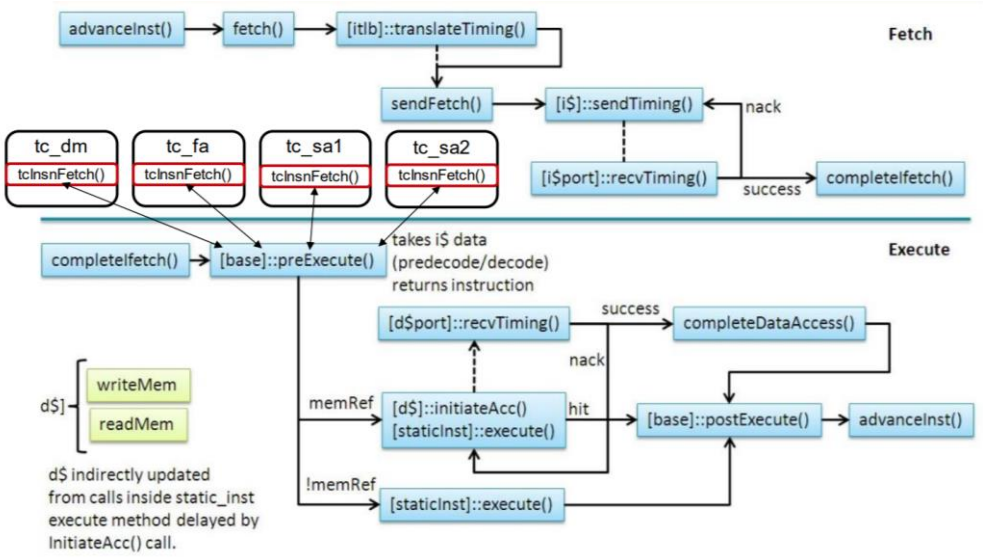

Figure 9: Changes to the TimingSimpleCPU for support of the trace cache. Modified image from [9].

5.2.3 Realistic Implementation

The previous sections on the gem5 integration discussed the changes made to the gem5 simulator with a small number of trace cache configurations. Information was displayed in this manner for simplicity. In our actual implementation, 16 trace cache configurations we modeled. Figures 10 and 11 describe the 16 cache configurations simulated in this report.

```
traceCache *tc_dm_64;
traceCache *tc_fa_64;
traceCache *tc_sa1_64;
traceCache *tc_sa2_64;

traceCache *tc_dm_32;
traceCache *tc_fa_32;
traceCache *tc_sa1_32;
traceCache *tc_sa2_32;

traceCache *tc_dm_16;
traceCache *tc_fa_16;
traceCache *tc_sa1_16;
traceCache *tc_sa2_16;

traceCache *tc_dm_8;
traceCache *tc_fa_8;
traceCache *tc_sa1_8;
traceCache *tc_sa2_8;
```

```
//trace caches
tc_dm_64 = new traceCache(64,1,16,1);
tc_fa_64 = new traceCache(1,64,16,1);
tc_sa1_64 = new traceCache(32,2,16,1);
tc_sa2_64 = new traceCache(16,4,16,1);

tc_dm_32 = new traceCache(32,1,16,1);
tc_fa_32 = new traceCache(1,32,16,1);
tc_sa1_32 = new traceCache(16,2,16,1);
tc_sa2_32 = new traceCache(8,4,16,1);

tc_dm_16 = new traceCache(16,1,16,1);
tc_fa_16 = new traceCache(1,16,16,1);
tc_sa1_16 = new traceCache(8,2,16,1);
tc_sa2_16 = new traceCache(4,4,16,1);

tc_dm_8 = new traceCache(8,1,16,1);
tc_fa_8 = new traceCache(1,8,16,1);
tc_sa1_8 = new traceCache(4,2,16,1);
tc_sa2_8 = new traceCache(2,4,16,1);
```

Figure 10: 16 trace cache configurations are instantiated and their respective pointer are made accessible within the BaseSimpleCPU class.

```
// send this request to all trace caches
tc_dm_64->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_fa_64->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa1_64->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa2_64->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);

tc_dm_32->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_fa_32->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa1_32->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa2_32->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);

tc_dm_16->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_fa_16->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa1_16->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa2_16->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);

tc_dm_8->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_fa_8->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa1_8->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
tc_sa2_8->tcInsnFetch(pkt->getAddr(), curStaticInst->isControl(), predictTakenSave);
```

Figure 11: 16 trace caches are accessed during the preExecute() function within the BaseSimpleCPU class.

6 Challenges

Our implementation faced challenges mainly focusing on getting the gem5 environment to run the SPEC benchmarks and integrating our model into the gem5 environment.

6.1 Integrating trace cache with gem5

We had initially planned to design the trace cache as a standalone gem5 ‘SimObject’ which could be used with any gem5 capable CPU. gem5 uses a packet-based protocol to communicate between the memory subsystem and the

CPU. Each instruction fetched from the memory is wrapped in a “Packet” class. The Packet contains all relevant information regarding the instruction such as the OPCODE, data fields, etc. To facilitate the use of this protocol, handshaking must be established between the CPU and cache to enable proper functionality. We were unable to implement our trace cache to handle the handshaking without affecting the functionality of the CPU. Hence, we implemented a snooping-based trace cache which would monitor the fetched packets (instructions) by the CPU.

6.2 Running SPEC06 Benchmarks

Since our changes were directly implemented in the TimingSimpleCPU, we were hesitant to rebuild gem5 which was shared across various user groups. We decided to copy over the contents of the complete gem5 directory and rebuild it to avoid any clashes with other users. We were able to gather statistics from certain benchmarks for our reference system without a trace cache but due to resource sharing conflicts, the scripts were not able to run after performing changes even after extensive debugging and stepping through each script.

We decided to build new scripts, which would just run the binaries without any checkpointing, which we sourced from the *‘/p/prometheus/private/adarsh1/spec2006_apps/’* directory. But since we did not have input files to the benchmarks readily available, we were able to run only a subset of the SPECCPU06 benchmarks. Some of the benchmarks required input files which were not readily available for us to pass as arguments to the benchmarks. This led to erroneous values for some of our benchmarks. Certain benchmarks require text/command inputs which were passed according to the details provided on SPECCPU06 documentation.

6.3 Exploring Other gem5 CPUs

We had originally planned to implement trace cache in various configurations of which we wanted to use the DerivO3 CPU provided by gem5. DerivO3 is an Out-of-Order CPU and we wanted to understand its interaction with trace cache. But due to the complexity faced while implementing the trace cache with the In-order CPU we had to discontinue the idea of implementing it. Rather we focused on generating more data from our implementation.

7 Results

7.1 Benchmark Data

We ran the gapbs benchmarks on the TimingSimpleCPU with the trace cache. We observe the number of trace cache hits and misses. We ran the system for 100 million instructions and the data are captured below. All these benchmarks were run using the ‘-g 20 -n 1’ option. The trace cache size is calculated as,

trace cache size = No. of Sets * Associativity

Associativity(n)	Size	#Misses	#Hits	#Sets
Direct Mapped (n=1)	32	439	3415	32
Set Associative (n=2)	32	78	3776	16
Set Associative (n=4)	32	57	3797	8
Set Associative (n=8)	32	29	3825	4
Fully Associative (n=32)	32	24	3830	1
Direct Mapped (n=1)	16	262	3592	16
Set Associative (n=2)	16	242	3612	8
Set Associative (n=4)	16	55	3799	4
Set Associative (n=8)	16	35	3819	2
Fully Associative (n=16)	16	25	3829	1
Direct Mapped (n=1)	8	392	3462	8
Set Associative (n=2)	8	243	3611	4
Set Associative (n=4)	8	47	3807	2
Fully Associative (n=8)	8	37	3817	1

Table 1 : Trace Cache with bfs benchmark (I$ Miss Rate: 0.00091)

Associativity(n)	Size	#Misses	#Hits	#Sets
Direct Mapped (n=1)	32	456	3444	32
Set Associative (n=2)	32	83	3817	16
Set Associative (n=4)	32	59	3841	8
Set Associative (n=8)	32	29	3871	4
Fully Associative (n=32)	32	26	3874	1
Direct Mapped (n=1)	16	315	3585	16
Set Associative (n=2)	16	256	3644	8
Set Associative (n=4)	16	50	3850	4
Set Associative (n=8)	16	36	3864	2
Fully Associative (n=16)	16	26	3874	1
Direct Mapped (n=1)	8	436	3464	8
Set Associative (n=2)	8	252	3648	4
Set Associative (n=4)	8	45	3855	2
Fully Associative (n=8)	8	28	3872	1

Table 2 : Trace Cache with bc benchmark (I$ Miss Rate: 0.00096)

Associativity(n)	Size	#Misses	#Hits	#Sets
Direct Mapped (n=1)	32	416	3484	32
Set Associative (n=2)	32	76	3824	16
Set Associative (n=4)	32	56	3844	8
Set Associative (n=8)	32	29	3871	4
Fully Associative (n=32)	32	26	3874	1
Direct Mapped (n=1)	16	289	3611	16
Set Associative (n=2)	16	265	3635	8
Set Associative (n=4)	16	46	3854	4
Set Associative (n=8)	16	27	3873	2
Fully Associative (n=16)	16	26	3874	1
Direct Mapped (n=1)	8	431	3469	8
Set Associative (n=2)	8	254	3646	4
Set Associative (n=4)	8	46	3854	2
Fully Associative (n=8)	8	27	3873	1

Table 3 : Trace Cache with sssp benchmark (I$ Miss Rate: 0.0001)

From these results you can see the number of trace cache hits for instantiation is high, especially compared to the number of misses, implying there could be potential for a performance increase if implemented to work within the CPU rather than just snooping from the CPU. You can also notice here that the direct mapped cache layout tends to have the worst performance.

We ran four SPEC06 benchmarks (sjeng, lbm, gobmk, milc) for 100 million instructions. The results are shown below in Table 4, Table 5, Table 6, Table 7.

Associativity(n)	Size	#Misses	#Hits	#Sets
Direct Mapped (n=1)	64	46	681	64
Set Associative (n=2)	64	29	698	32
Set Associative (n=4)	64	6	721	16
Fully Associative (n=64)	64	6	721	1
Direct Mapped (n=1)	32	132	595	32
Set Associative (n=2)	32	44	683	16
Set Associative (n=4)	32	6	721	8
Fully Associative (n=32)	32	6	721	1
Direct Mapped (n=1)	16	133	594	16
Set Associative (n=2)	16	39	688	8
Set Associative (n=4)	16	6	721	4
Fully Associative (n=16)	16	7	720	1
Direct Mapped (n=1)	8	128	599	8
Set Associative (n=2)	8	37	690	4
Set Associative (n=4)	8	11	716	2
Fully Associative (n=8)	8	6	721	1

Table 4: Trace Cache with sjeng benchmark (I$ Miss Rate : 0.000013)

Associativity(n)	Size	#Misses	#Hits	#Sets
Direct Mapped (n=1)	64	66	190	64
Set Associative (n=2)	64	32	218	32
Set Associative (n=4)	64	11	245	16
Fully Associative (n=64)	64	11	245	1
Direct Mapped (n=1)	32	35	221	32
Set Associative (n=2)	32	15	241	16
Set Associative (n=4)	32	13	243	8
Fully Associative (n=32)	32	11	245	1
Direct Mapped (n=1)	16	15	241	16
Set Associative (n=2)	16	13	243	8
Set Associative (n=4)	16	11	245	4
Fully Associative (n=16)	16	11	245	1
Direct Mapped (n=1)	8	56	200	8
Set Associative (n=2)	8	15	241	4
Set Associative (n=4)	8	12	244	2
Fully Associative (n=8)	8	12	244	1

Table 5 : Trace Cache with lbm benchmark (I$ Miss Rate : 0.000007)

Associativity(n)	Size	#Misses	#Hits	#Sets
Direct Mapped (n=1)	64	6	1345	64
Set Associative (n=2)	64	6	1345	32
Set Associative (n=4)	64	6	1345	16
Fully Associative (n=64)	64	6	1345	1
Direct Mapped (n=1)	32	6	1345	32
Set Associative (n=2)	32	7	1344	16
Set Associative (n=4)	32	7	1344	8
Fully Associative (n=32)	32	6	1345	1
Direct Mapped (n=1)	16	305	1046	16
Set Associative (n=2)	16	48	1303	8
Set Associative (n=4)	16	11	1340	4
Fully Associative (n=16)	16	7	1344	1
Direct Mapped (n=1)	8	136	1215	8
Set Associative (n=2)	8	51	1300	4
Set Associative (n=4)	8	10	1341	2
Fully Associative (n=8)	8	6	1345	1

Table 6: Trace Cache with gobmk benchmark (I$ Miss Rate : 0.000024)

Associativity(n)	Size	#Misses	#Hits	#Sets
Direct Mapped (n=1)	64	88	989034	64
Set Associative (n=2)	64	42	989080	32
Set Associative (n=4)	64	19	989103	16
Fully Associative (n=64)	64	17	989105	1
Direct Mapped (n=1)	32	70	989052	32
Set Associative (n=2)	32	60	989062	16
Set Associative (n=4)	32	25	989097	8
Fully Associative (n=32)	32	17	989105	1
Direct Mapped (n=1)	16	67	989055	16
Set Associative (n=2)	16	54	989068	8
Set Associative (n=4)	16	23	989099	4
Fully Associative (n=16)	16	17	989105	1
Direct Mapped (n=1)	8	118	989004	8
Set Associative (n=2)	8	58	989064	4
Set Associative (n=4)	8	28	989094	2
Fully Associative (n=8)	8	22	989100	1

Table 7: Trace Cache with milc benchmark (I$ Miss Rate : 0.00001)

From these spec06 benchmarks, the same conclusions can be drawn as the gapbs. The fully associative instantiations tend to have the lowest number of misses, and therefore also the higher number of hits.

7.2 Trace Cache Trends

We saw one major reccuring trend among all the benchmarks that were performed. As seen in Figure 12, with the increase in associativity we saw a decrease in the number of misses per trace cache. This was common across all the

various sizes of the trace cache. We also observed that the trace cache with lower size showed higher amounts of misses when compared to cache sizes with higher sizes for the same associativity.

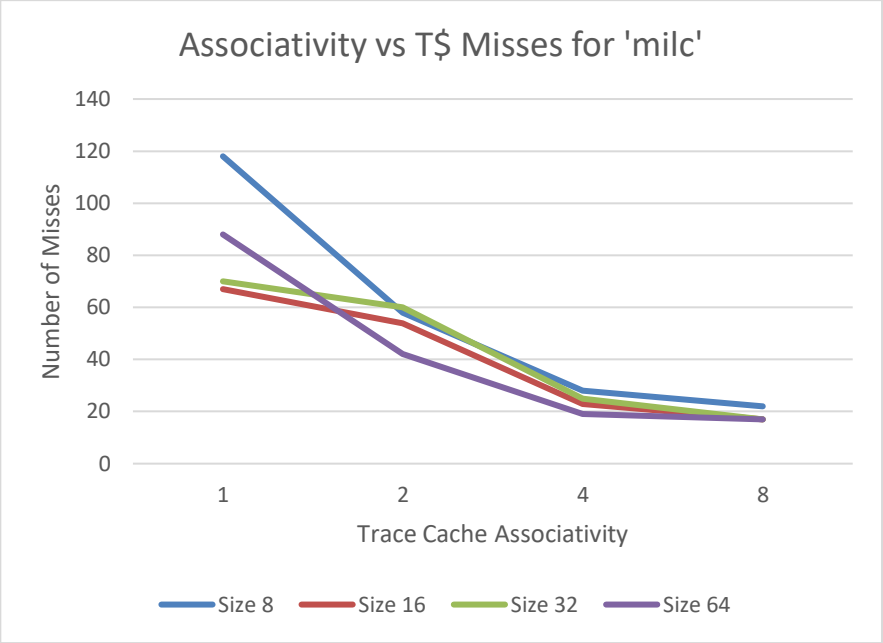

Figure 12: Trace cache misses plotted vs the associativity of the trace cache for various trace cache sizes.

While collecting benchmark data we output what the final cache block looked like for a given object, i.e. size, associativity, and number of sets, and found the caches that were most likely to underperform were the size {32,16,8} direct mapped caches. Each with less lines stored compared to the other instaniations. The most utalized tended to be size {64,32,16} fully associative caches. This supports the expected results. This also shows that for a given program, size 64 might be too much, and since the trace cache encourages fast accesability, further tests should be ran to determine how much smaller the cache can go while still having the ability to increase performance.

8 Conclusions

With reduced transistor tech nodes in this modern era, we have more space in the SoC to accommodate for a greater number of transistors per single core. Therefore, trace cache may now be a viable solution again for performance gains due to Icache miss rate being a bottleneck in a system. Using our simple trace cache design that was developed to snoop on the fetching unit in the TimingSimpleCPU, we were able to conclude:

- when we increase the associativity in trace cache, the number of trace cache misses decreases, this leads us to believe that a trace cache with larger order associativity improves the performance of the system.

- also, when we decrease the size of the trace cache, the number of misses increases.

From these observations, we can make a decision about what size and associativity would be a perfect fit for the trace cache depending on the area budget we have in the SoC. We also observed that of the filled trace cache lines they are fully utilized i.e., the total number instructions filled in these trace cache lines are 16 while some of the trace cache lines were empty.

Future work to be done in this area includes but is not limited to: Implementing a full model within gem5 rather than a simple testing model, adding a trace predictor and/or victim cache to analyze its use, adding different replacement policies to analyze whether they are effective for trace caches.

References

[1] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, Paris, France, 1996, pp. 24–34. doi: 10.1109/MICRO.1996.566447.

[2] E. Rotenberg, S. Bennett, and J. E. Smith, "A trace cache microarchitecture and evaluation," *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 111–120, Feb. 1999, doi: 10.1109/12.752652.

[3] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, doi: 10.1145/2024716.2024718.

[4] R. Rosner, A. Mendelson, and R. Ronen, "Filtering techniques to improve trace-cache efficiency," *Proc. 2001 Int. Conf. Parallel Archit. Compil. Tech.*, 2001, doi: 10.1109/PACT.2001.953286.

[5] chustedde, *Filtered Trace Cache using SimpleScalar*. 2018. Accessed: May 09, 2022. [Online]. Available: https://github.com/chustedde/simplescalar-tracecache

[6] M. Postiff, G. Tyson, and T. Mudge, "Performance Limits of Trace Caches," *J. Instr.-Level Parallelism*, vol. 1, 1999.

[7] "Tutorial: Easily Running SPEC CPU2006 Benchmarks in the gem5 Simulator | Mark Gottscho's Blog." https://markgottscho.wordpress.com/2014/09/20/tutorial-easily-running-spec-cpu2006-benchmarks-in-the-gem5-simulator/ (accessed May 09, 2022).

[8] "traceCacheClass/traceCache.cc at main · ariwanski/traceCacheClass," *GitHub*. https://github.com/ariwanski/traceCacheClass (accessed May 09, 2022).

[9] "gem5: Simple CPU Models." https://www.gem5.org/documentation/general_docs/cpu_models/SimpleCPU (accessed May 09, 2022).