


# MBPlib: Modular Branch Prediction Library

Emilio Domínguez-Sánchez

Computer Engineering Department (DITEC)

University of Murcia

Murcia, Spain

emilio.dominguezs@um.es 

Alberto Ros

Computer Engineering Department (DITEC)

University of Murcia

Murcia, Spain

aros@ditec.um.es 

**Abstract**—Branch predictors are the hardware logic that tries to guess the outcome of a branch instruction before its execution. Currently, researchers make use of simulation tools to measure the accuracy of their predictors against hundreds of program traces. However, these simulations require multiple hours of computation time. This makes the prototyping slow and limits the ability of the researcher to test different strategies. Besides, current simulators are built as frameworks instead of libraries, in the sense that they call the user code and not the other way around. As a result, the user has no control of the program execution and they cannot optimize it for the experiment at hand.

In this paper we present *Modular Branch Prediction Library* (MBPlib), an open-source C++ library that solves the aforementioned issues. MBPlib runs over  $18.4 \times$  faster than the current fastest framework, and its trace format uses  $6.5 \times$  less disk space. MBPlib also makes development easier by providing utilities that are typically used as subcomponents in most branch prediction designs. Moreover, the library features one of the largest collections of example implementations, including traditional as well as state-of-the-art predictors.

MBPlib will allow researchers to significantly reduce the time needed for evaluation. Furthermore, by giving the option of obtaining results within seconds, as well as by means of the broad collection of examples, written in a modern and uniform code style, MBPlib can significantly decrease the barrier to entry into the field. Thus, we believe that MBPlib is also a great tool for computer architecture classes.

**Index Terms**—branch-prediction, simulation, library.

## I. INTRODUCTION

Branch predictors are the hardware logic that tries to guess the outcome of the branch instruction before its execution. Having an accurate branch predictor is critical to achieving high performance in modern pipelined architectures [1], [2]. As a matter of fact, experiments on real processors showed that the performance could be improved by 13% if the mispredictions were reduced by half [3].

Branch prediction designs are commonly evaluated using trace-based simulators. Microarchitecture-agnostic metrics, such as the mispredictions per kilo-instructions (MPKI), have become a de-facto standard in the field. Most recent research articles only provide the MPKI of their design, leaving performance improvements as a second-class metric [4]–[7]. Similarly, MPKI is the only metric used in the Championships in Branch Prediction [8]–[12].

One reason for the use of microarchitecture-agnostic metrics is that research has been influenced by the championships methodology, which needs to be based on simple tools to

make the competition more accessible. Another reason is that it has been shown that having fewer mispredictions per kilo-instructions is correlated with better performance. Plus, it is always preferable to have a metric that does not depend on the microarchitecture evaluated. Lastly, and perhaps more importantly given the continuous increase in the complexity of the predictors, the simulators that do not need to reproduce the processor internals or model non-branch instructions are orders of magnitude faster. This makes it possible to run more or longer traces.

Two popular trace-based simulators that model branch predictors and have a user-friendly interface are the Championship Simulator (ChampSim) [13] and the framework from the 5th Championship in Branch Prediction (CBP5) [12]. ChampSim is a cycle-accurate simulator for microarchitecture study. It implements an out-of-order processor and can measure the instructions per cycle as well as the MPKI. Thus, ChampSim traces include a detailed description of the execution. On the other hand, the CBP5 framework is micro-architecturally independent and its trace format only contains information about the program branches and the number of instructions. As a result, the CBP5 framework can be  $50 \times$  faster (section VII), making it the preferred option by a lot of researchers, even though it is not maintained.

In this paper we present the Modular Branch Prediction library (MBPlib). MBPlib is an open-source<sup>1</sup> C++ library for writing branch predictors implementations and performing microarchitecture-agnostic simulations. As a tool, MBPlib offers the following:

- 1) It is  $18.4 \times$  and  $923 \times$  faster than the CBP5 framework and ChampSim, respectively.
- 2) It uses a compact trace format. Our translation of two sets of traces use  $6.5 \times$  and  $42 \times$  less space, respectively.
- 3) It offers utilities to avoid reimplementing common functionality and it favors designs written with composability in mind, where predictors also become components.
- 4) It is built on top of modern technologies, such as C++17 and CMake, and uses JSON as the output format.
- 5) It is designed as a library instead of a framework, which means that the user code calls MBPlib and not the other way around, making more easy to use it in conjunction with other software.

<sup>1</sup>MBPlib is published at <https://github.com/useredsa/MBPlib/>.

## II. MOTIVATION

The relevance of branch prediction has only increased in the past years due to the continuous increase in the width of the pipeline and the number of ways of superscalar processors. The reason for this is that the deeper in the pipeline the evaluation of the branch is, the higher the number of penalty cycles incurred on a misprediction. And the higher the number of ways, the higher the percentage of time that the stalls suppose. For instance, a microarchitecture that fetches one instruction per cycle, evaluates branches in the 5-th cycle and has a branch predictor with 5 MPKI has a CPI of  $1 + 0.005 \times 4 = 1.02$ . If we could reduce in 1 the MPKI, the CPI would become  $1 + 0.004 \times 4 = 1.016$ , giving a speedup of  $\frac{1.02}{1.016} \approx 0.4\%$ . On the other hand, a microarchitecture that fetches 4 instructions per cycle and evaluates the branches in the 11-th cycle would achieve  $0.25 + 0.005 \times 10 = 0.3$  CPI and  $0.25 + 0.004 \times 10 = 0.29$  CPI, respectively. Hence, the speedup in this case would be  $\frac{0.3}{0.29} \approx 3.4\%$ .

In parallel to the increasing relevance of branch prediction, branch prediction schemes have also become more complex. If an old GShare [14] predictor had 3 parameters: the size of the table, the number of bits of each entry and the branch history length used to index it, a modern TAGE [15] predictor can have more than 50 parameters. Since at the very least it has the same 3 plus the number of bits used for the tag and utility counters for each of its tables, which can be 10 or more. Thus, finding the optimal parameter values requires testing more configurations.

Moreover, as years have passed, the evaluation has also become more thorough and the number of traces used in evaluation has increased. For instance, the CBP4 used a set of 40 traces, while the last CBP used 223 traces for training and 440 for evaluation [16]. Now it is common to use hundreds of traces from popular benchmark suites [12], [17]. Besides, sometimes the researchers use long traces to measure how the predictor adapts to changes in the program behavior. For example, also in the last CBP, 23 and 40 traces of the training and evaluation sets, respectively, contained more than 1 billion instructions.

In section VII we show that simulating a single trace of 100 million instructions in ChampSim, which models the whole processor, takes 11.7min on average. Thus, simulating the whole training set of traces of the CBP5 in ChampSim would require roughly 84h of computational time. This justifies the need for fast microarchitecture-agnostic simulators that can be used to explore the design space, tune the predictor parameters and run very long traces.

### A. Objectives

The 5th Championship Branch Prediction (CBP5) [12] was held in 2016. The organization committee provided a microarchitecture-agnostic common evaluation framework and a set of 223 training traces. To create the set, they obtained thousands of traces from programs of different popular software suites and followed a complex classification process

based on principal component analysis and clustering techniques to select a representative set [16]. Obviously, this process is time-consuming, and for that reason the traces from the CBP5 were still used after the competition [5]. However, the CBP5 framework is not maintained, and 7 years have passed since the championship celebration.

Currently, there is no open-source simulator that is as easy to use as those featured in the championships. MBPlib tries to fill the gap by being a simulator with those two characteristics. However, since MBPlib is being designed in 2022, we are also expected to improve upon previous simulators.

Our primary goal was to make MBPlib as fast as possible. Even with the CBP5 framework, it is necessary to wait multiple minutes to obtain the results for a small number of traces, not to mention for all 223 traces. Students and beginning researchers often need to see the effect of small changes in the predictor's performance to understand which strategies work. Thus, MBPlib should make prototyping in real-time possible. That is, to obtain results within seconds. In order to be faster, MBPlib had to leave behind the CBP5 trace format, which is encoded in plain text, and switch to a binary format. Still, we believed that in order to gain broad adoption, it was essential to offer a curated set of traces. For that reason, we translated the CBP5 traces to our new format.

The second objective of MBPlib was to integrate with other projects. Frameworks that come with their own build system, typically a Makefile file, can make it difficult to use third-party code. There is no way to manage the user dependencies and there is no control over the program's start-up. It becomes necessary to create scripts that invoke the framework's compilation process. To cope with this, we conceived MBPlib as a simulation library instead of a framework. MBPlib integrates inside the user project and not the other way around.

In a nutshell, while prioritizing the objective of being fast, MBPlib shall be modular, easy to use and accessible to a broad public.

## III. ARCHITECTURE OF MBPLIB

We start with a description of MBPlib, detailing its architecture as a software suite. Rather than as a single library, MBPlib is built as three C++17 libraries that can be used independently. They are the simulation, utilities and examples libraries, targeted towards simulation, development and education, respectively.

The *simulation library* is what would correspond to the whole framework in other software suites. Basically, it offers all the routines needed to run a user-defined branch predictor for a program trace and obtain a JSON object with the simulation results. Besides, the simulation library offers the trace reader and trace writer as subcomponents. What this means is that the user can also link a library that contains only the trace reader or writer, which is useful for creating applications that inspect or modify the program traces.

The *utilities library* offers software implementations of components that are present in most branch predictors, such as fixed-width saturated counters or a class that maintains a hash

of the global branch history. These components avoid the need to reimplement common functionality in different predictors and can be better than the short pieces of code that are typically written to replace them, because they handle all inputs, include error checks and can be optimized by the software maintainers. For instance, by modeling the fixed-width counters as a class we can create custom arithmetical operators for it, providing a simple and modern interface. (Note that even though the utilities library is part of MBPlib, it can still be used to implement branch predictors in other simulators.)

Finally, the *examples library* offers implementations for the most well-known branch predictors. Including state-of-the-art predictors, like BATAGE [5], a recent successor to the famous predictor TAGE [15]. These implementations can serve two purposes. One is to learn and teach about branch prediction techniques or how to take advantage of the utilities library. The other is to use them as subcomponents of another predictor. For instance, a lot of predictors use a bimodal predictor [18] as a subcomponent. Others combine the result obtained by different prediction strategies.

#### IV. THE SIMULATION LIBRARY

The simulation library, which is abbreviated as the simulator throughout the text, is a lightweight library focused on obtaining microarchitecture-agnostic metrics. In simple words, what the simulator does is read a program trace that contains the branches seen during the execution, ask the branch predictor to anticipate the outcome of those branches, and record how many times the predictor was incorrect.

Our simulator uses a custom trace format named *Simple Binary Branch Trace (SBBT)*. The format is greatly inspired by the CBP5 framework trace format, BT9, but it is designed to take less space and allow faster parsing. The BT9 format is a plain-text format that starts by describing a graph where the nodes are the branches present in a program and their possible outcomes are the edges and then follows with a section that describes the sequence of edges taken. The SBBT format, on the other hand, has only a small header without the graph description and the rest is just a concatenation of packets of 128 bits. Each of these packets includes a description of the current branch and its outcome. In addition, SBBT is a binary format. Hence, it uses less space and saves the time required for parsing the text format.

It is common practice to distribute the traces compressed and let the simulator decompress them. MBPlib can decompress traces compressed with `xz`, `gzip`, `lz4` or `zstd`. Of all the available compression algorithms, we used `zstandard` (`zstd`) [19] because it had the best decompression speed for the SBBT format. Moreover, in the case of `zstandard`, we saw that a bigger compression factor did not make the decompression slower. Thus, we use the biggest compression ratio available (level 22).

In practice, the absence of the branch graph in the header makes the SBBT traces contain more redundant information. This may make the files bigger but it makes the simulator avoid the cache misses from accessing a big hashed structure to

read the branch metadata. Using a good compression method also helps to reduce the amount of redundant information. The CBP5 training traces in SBBT format take 769 MB when compressed using `zstandard` [19], while the same traces in BT9 format, when compressed with the same method, take only 504 MB. However, the original CBP5 framework used `gzip` [20] to compress its traces and the traces distributed used 5.4 GB of disk space. That makes the new set of traces  $7.3 \times$  smaller. Besides, 700 MB to 800 MB is already an affordable disk space. Consequently, we considered more important the simulation speed.

Throughout the rest of the section we describe the simulator interface, the SBBT format, and the output produced by the simulator.

##### A. Predictor Interface

In MBPlib, a branch predictor is a class that inherits from `mbp::Predictor` and overrides the following three functions.

- `bool predict(uint64_t)`  
Obtains the outcome prediction for a given instruction address. This function shall not modify the state of the predictor in any way that would affect future predictions.
- `void train(const mbp::Branch&)`  
Updates the predictor considering what was the outcome of the branch in the current scenario.
- `void track(const mbp::Branch&)`  
Updates the predictor scenario based on the outcome of the given branch. The word *scenario* refers to the information stored about the recent program behavior, such as the outcome of recent branches.

The predictor class also includes other functions which can be optionally overridden, though for the sake of brevity, we do not describe them here, but in the software manual. They can be used, for example, to include predictor data in the simulator's output.

##### B. Train and Track

In contrast to ChampSim and the CBP5 framework, MBPlib has two functions that update the predictor. This distinction was done with composability in mind.

Branch predictors have two types of data structures. Data structures of the first type determine the prediction given the branch address and the current scenario, while data structures of the second type record the current scenario. Thus, even though both types of structures condition what the output of calling `predict` is, the difference is that the information contained in the structures of the second type is used as input to access the structures of the first type whereas the information retrieved from the structures of the first type is used to decide the prediction. Thus, it is as if the structures of the second type were also parameters to `predict`. But they cannot be parameters of `predict` because each predictor decides what information to store for a prediction.

If the predictor is being called by the simulator then `track` is invoked for all branches while `train` is invoked before `track`

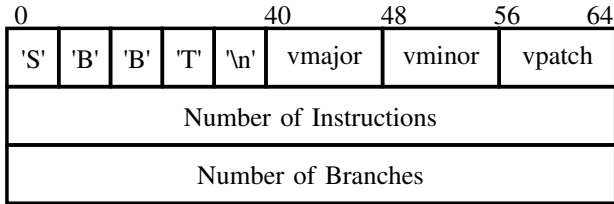


Fig. 1: SBBT header, spanning 196 bits.

Trace Set	Num. of Traces	Original Size	Translated Size	Size Ratio
CBP5 - Training	223	5.4 GB	760 MB	7.3 ×
CBP5 - Evaluation	440	4.0 GB	727 MB	5.0 ×
DPC3	95	30.0 GB	727 MB	42.0 ×

TABLE I: Size reduction of the set of traces translated.

but only for conditional branches. However, when the predictor is part of a bigger (meta-)predictor or we place a branch filter before it, it will be the owning component the one deciding which functions are called. For example, a meta-predictor may be interested in training a subcomponent only when there is no other component giving the correct prediction, which is called a partial update policy. But the meta-predictor still needs to call the `track` function, because knowing the outcome of the current branch may be necessary to predict other branches. Similarly, a filter may decide that it is not necessary to track some branches.

### C. Trace Format

The SBBT format, version 1.0.0, is a concatenation of packets of 128 bits that is preceded by a header of 192 bits.

The SBBT header structure is shown in fig. 1. The first 5 bytes of the header is the signature that characterizes the SBBT filetype, which corresponds to the string "SBBT\n" if read as plain text. The following 3 bytes encode the major (1), minor (0) and patch (0) version numbers as three unsigned 8 bit numbers. The rest of the header are two 64 bits numbers, encoded in little endian: the number of instructions (branch and non-branch) excuted by the processor during the tracing procedure and the number of branches present in the trace.

Each SBBT packet of 128 bits represents a branch instruction, as depicted in fig. 2. The packet is divided into two blocks of 64 bits, which are encoded in little endian. The first 64-bit block contains the branch opcode, the outcome and the virtual address of the branch instruction. The second 64-bit block contains the number of instructions that got executed since the last branch (without counting either branch) and the virtual address of the branch target. The format stores the number of instructions between branches to know the instruction number of each branch. This allows the user to run only the first  $n$  instructions of the trace and to use certain amount of instructions as warm-up: instructions whose mispredictions are not counted.

The addresses are encoded using the 52 most significant bits of each 64-bit block. With this width we can encode both the

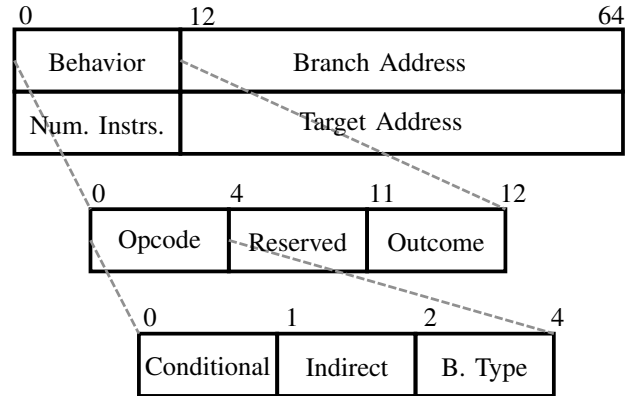


Fig. 2: SBBT branch packet, spanning 128 bits.

48-bit virtual addresses used by the x86-64 architecture [21], [22] and the 52-bit virtual addresses used by the ARMv8-A architecture with the Large Virtual Address Extension [23]. The 64-bit addresses used in the simulator are obtained by performing an arithmetical shift of 12 bits.

The other 12 bits of the first block are used to encode the branch behavior. The first 4 encode the branch opcode. The next 7 bits that are reserved for future use and the last bit indicates the branch outcome. The opcode follows closely the definition of opcode used in the BT9 traces. The first two bits indicate whether the branch is conditional and indirect, respectively. The next two the base type of the branch, which can be JUMP (00), CALL (10) or RET (01). With this notation, branches that push or pop from the return address stack are labeled as CALL or RET, respectively. The rest of the branches are labeled as JUMP.

In the second block, the additional 12 bits encode the number of instructions executed in the path to this branch. Hence, there can be at most 4096 instructions separating two consecutive branches. This is not a small number, since approximately 15%to25% of all the instructions are branches [24], and traces with a small number of branches are of little relevance in branch prediction. In particular, none of the 663 traces from the CBP5 and the 95 traces [25] from the 3rd Data Prefetching Championship [17], which are based on the SPEC17 suite [26], have two consecutive branches spread more than 4096 instructions apart.

Lastly, not all combinations of values are valid. Two rules should be obeyed. First, if the branch is not conditional, the outcome bit must always mark that the branch was taken. Second, when the branch is conditional and indirect and the outcome bit indicates that it is not taken, the target virtual address must be null (0x0).

### D. Available Traces

MBPlib comes with the traces of the Championship Branch Prediction 5 (CBP5) and the traces of the 3rd Data Prefetching Championship (DPC3), which are linked in ChampSim's GitHub repository [27]. Table I shows the number of traces

present in each set and their total size. All of the sets of SBBT traces occupy less than 800 MB. This means a reduction of  $6.5\times$  in disk space on average for the totality of the CBP5 traces (train and evaluation). The translated DPC3 traces are  $42\times$  smaller, but in this case the comparison is not fair because ChampSim needs to store the registers accessed by the instructions and information about all types of instructions, not just branches.

The programs used to translate traces from the BT9 and champsimtrace formats are linked in MBPlib's repository. Therefore, the user can translate any traces that they had already recorded for both simulators. Apart from that, we also provide an instrumentation module to trace an x86 executable. This module is to be used with Intel's PIN tool [28].

### E. Output Format

MBPlib embeds user data into the output. Both static data the user wants to add to identify details of the simulation and dynamic data recorded during the execution of the trace. Hence, we had to choose an output data type that was flexible enough. At the same time, we also wanted a machine-friendly output format that could be easily parsed by other tools and scripting languages afterward. Therefore, we decided to make the simulator return a JSON object.

Listing 1: Example output.

```

1 {
2   "metadata": {
3     "simulator": "MBPlib std simulator",
4     "version": "v0.5.0",
5     "trace": "traces/SHORT_SERVER-1.sbbt.zst",
6     "warmup_instr": 0,
7     "simulation_instr": 1283944652,
8     "exhausted_trace": true,
9     "num_conditional_branches": 162876464,
10    "num_branch_instructions": 16056,
11    "predictor": {
12      "name": "MBPlib GShare",
13      "history_length": 25,
14      "log_table_size": 18,
15      "track_only_conditional": false
16    }
17  },
18  "metrics": {
19    "mpki": 3.312043080187229,
20    "mispredictions": 4252480,
21    "accuracy": 0.973891378192002,
22    "num_most_failed_branches": 36,
23    "simulation_time": 8.443604087
24  },
25  "predictor_statistics": {},
26  "most_failed": [
27    {
28      "ip": 1995000000,
29      "occurrences": 3231824,
30      "mpki": 0.22369422198426667,
31      "accuracy": 0.9111303709607949
32    },
33    {
34      "ip": 2148302608,
35      "occurrences": 1638183,
36      "mpki": 0.19968617774966207,
37      "accuracy": 0.8434936756149954
38    },
39    ...
40  ]
41 }

```

JSON is well-known, flexible, human friendly and quickly parseable. Furthermore, it is also straightforward for the users to encode any type of data as a JSON object. The JSON library used is nlohmann/json [29], a JSON library for modern C++ with intuitive syntax and trivial integration.

An example JSON output from MBPlib can be seen in listing 1. The metadata section includes the type of simulator used and its version, the trace read, the number of warm-up instructions, the number of simulation instructions, the number of branch instructions and the number of conditional branches. The metrics section tells the user the number of mispredictions, the MPKI, the accuracy, the simulation time and the minimum number of branches that account (on their own) for half of the mispredictions. The most\_failed section lists these branches.

The sections including user output are predictor, inside metadata, and predictor\_statistics. Typically, in the first section we want to include the name of the predictor and the selection of parameters which was used to configure it. For example, from the information included in listing 1 we can tell that this is a 64 kB version of GShare because it is using a table of  $2^{18}$  two-bit counters. On the other hand, in the second section we want to include execution statistics that will allow us to gather information unique to our design. For example, a measure of the number of conflicts inside the predictor tables.

## V. THE UTILITIES AND EXAMPLES LIBRARIES

The utilities library offers common logic extracted from the five championships in branch prediction. This reduces the amount of repeated code, offers a good deal of abstraction and achieves an homogeneous design among branch predictors.

Listing 2: GShare Implementation with MBP utilities.

```

1 template <int H = 15, int T = 17>
2 struct Gshare : mbp::Predictor {
3   std::array<mbp::i2, (1 << T)> table;
4   std::bitset<H> ghist;
5
6   uint64_t hash(uint64_t ip) const {
7     return mbp::XorFold(ip ^ ghist.to_ullong(), T);
8   }
9   bool predict(uint64_t ip) override {
10    return table[hash(ip)] >= 0;
11  }
12  void train(const mbp::Branch& b) override {
13    table[hash(b.ip())].sumOrSub(b.isTaken());
14  }
15  void track(const mbp::Branch& b) override {
16    ghist <=< 1;
17    ghist[0] = b.isTaken();
18  }
19 };

```

In a sense, the biggest achievement is that the utilities library makes it easy to build branch predictors by gluing together different components. For example, if we use the class bitset from the standard C++ library to store the branch history and i2 and XorFold from MBPlib, which are a two-bit saturating signed counter and a function to compute the hash of the

branch address and history register, a GShare predictor [14] can fit in barely 20 lines of code (see listing 2).

Initially, we included the examples library to showcase how the utilities library could help and the implementations were built specifically for MBPlib. As a byproduct of this, all of them share a similar coding style. At the same time, the use of components makes our implementations noticeably shorter than those found online. The most remarkable example of this would be the TAGE predictor [15], which is implemented with barely 150 lines of code in MBPlib, in contrast to the 700 lines of code of the championship version.

Moreover, as with the GShare predictor above, all of the examples allow the user to tweak the predictor parameters. For example, one can create a TAGE predictor with an arbitrary number of tables, each using its own history length and counter and tag width. And, of course, the proper function is overridden so that these options get recorded in the metadata of the predictor and then appear in the simulator’s output.

Due to this, the examples library could serve a teacher to set up an exercise in which the students measure how the MPKI varies with respect to some parameters or try to find the best set of parameters. All in all, we reckon that the utilities library is a fundamental addition to the suite and that the examples library is a valuable learning resource.

## VI. SAMPLE USE CASES

A software suite that wants to find a place in the research ecosystem must consider which common usage scenarios it will handle. When it comes to branch prediction, we identified three common use cases: parameter optimization, predictor comparison and composition. In the case of parameter optimization, the set of parameters could be known beforehand or we could perform a search in the design space based on simulation results.

### A. Parameter Optimization

Consider the GShare implementation shown in listing 2. The predictor depends on two parameters, the size of the table (controlled with parameter  $T$ ) and the length of the global history register ( $H$ ). The parameter  $T$  may be fixed based on the available budget, but we would like to choose the value of  $H$  that gives the best results. To measure the effect of some parameters on the mispredictions, we would like to generate multiple simulation executables for the different sets of values. In addition, it can be helpful to compile the different versions of our predictor separately to let the compiler perform compile-time optimizations based on the values of the parameters.

Frameworks such as the CBP5 and ChampSim include their own build system created to produce a single executable. In the case of ChampSim, it even needs to run a configuration phase before the compilation. Due to this, the user needs to create scripts to invoke the build system or adapt it to generate multiple executables.

MBPlib, on the other hand, is designed as a library and makes the user responsible for creating the binaries. For the

example we had at hand, the simplest solution would be to use a CMake for-loop to define multiple executables with similar characteristics, as shown in listing 3.

Listing 3: Generating Parametrized Executables.

```

1 foreach(h RANGE 6 30)
2   add_executable(gshare_${h}_64KB src/main.cpp)
3   target_link_libraries(gshare_${h}_64KB
4     PRIVATE mbp_sim mbp_utils)
5   target_compile_definitions(gshare_${h}_64KB
6     PRIVATE "PREDICTOR=Gshare<${h}, 18>")
7 endforeach()

```

### B. Searching the Parameter Space

The generation of multiple executable files works well when the number of parameters is small or when we want to see the effect of only a subset of them. However, the state-of-the-art predictors, in particular the winners of the CBPs, have dozens of parameters. In that case, we cannot afford to simulate all possible combinations, because the amount is exponential in the number of parameters.

Possible solutions include using genetic algorithms, machine learning or Bayesian optimization to find the best set of parameters. MBPlib does not try to be a library that can do all of these, since there are existing C++ libraries for these purposes. However, once more, the fact that MBPlib is built as a library helps in this regard. The user also has complete control of the program execution. Thus, they can integrate other libraries in their code and call MBPlib as part of the optimization process.

### C. Predictor Comparison

Another typical use case is to compare the effectiveness of adding a new component, like a loop predictor, to our design. Apart from the standard simulator, MBPlib offers a comparison simulator that simulates two predictors in parallel. This simulator can be used to determine which occurrences are mispredicted by only one of them. In this simulator, the `most_failed` section of the output contains the branches which accounted for the biggest difference in MPKI. Thus, it can serve to tell which branches get predicted better and if there are some whose predictability worsens.

### D. Reusability and Composability

Having two separate functions (`train` and `track`) to update the different data structures is very useful to improve reusability and composability. We show why with a practical example.

The original tournament predictor [30] had a bimodal and a GShare predictor internally. The rationale for using two branch predictors was that, while a GShare predictor becomes more accurate after learning the program patterns, it requires more training time to adapt to new behaviors. What the tournament predictor does is choosing between the predictions of the bimodal and the GShare components based on the accuracy that they show at the moment. Thus, the tournament predictor is a meta-predictor whose outcome bit, instead of the outcome

of the branch, tries to guess which prediction must be chosen. A generalization of this idea is to use two arbitrary predictors as base components and another one as a meta-predictor.

To implement these types of generalizations, in which we need to use another predictor, we can add a member of type `mbp::Predictor` to our class. The separation of the `train` and `track` functions creates the opportunity of calling just one of these (even if the branch is conditional) or to call them with different values.

Listing 4: Generalized Tournament Predictor.

```

1 struct TournamentPred : mbp::Predictor {
2   std::unique_ptr<mbp::Predictor> meta;
3   std::unique_ptr<mbp::Predictor> bp0;
4   std::unique_ptr<mbp::Predictor> bp1;
5   // Cached Data
6   uint64_t predictedIp;
7   bool tracked;
8   bool provider;
9   std::array<bool, 2> prediction;
10
11 TournamentPred(std::unique_ptr<Predictor> meta,
12                std::unique_ptr<Predictor> bp0, std:::
13                unique_ptr<Predictor> bp1)
14   : meta(std::move(meta)),
15     bp0(std::move(bp0)),
16     bp1(std::move(bp1)),
17     tracked(true) {}
18
19 bool predict(uint64_t ip) override {
20   if (predictedIp == ip && tracked == false)
21     return prediction[provider];
22   predictedIp = ip;
23   tracked = false;
24   provider = meta->predict(ip);
25   prediction[0] = bp0->predict(ip);
26   prediction[1] = bp1->predict(ip);
27   return prediction[provider];
28 }
29
30 void train(const mbp::Branch& b) override {
31   this->predict(b.ip());
32   bp0->train(b);
33   bp1->train(b);
34   if (prediction[0] != prediction[1]) {
35     mbp::Branch metaBranch = {
36       b.ip(), b.target(), b.opcode(),
37       prediction[1] == b.isTaken();
38     };
39     meta->train(metaBranch);
40   }
41 }
42
43 void track(const mbp::Branch& b) override {
44   meta->track(b);
45   bp0->track(b);
46   bp1->track(b);
47   tracked = true;
48 }
49
50 json metadata_stats() const override {
51   return {
52     {"name", "MBPlib Tournament"},
53     {"metapredictor", meta->metadata_stats()},
54     {"predictor_0", bp0->metadata_stats()},
55     {"predictor_1", bp1->metadata_stats()};
56 };

```

For example, the implementation for the generalized tournament predictor, illustrated in listing 4, only trains the meta-predictor when the output from the base predictors is different (line 33). And in that case, instead of training the predictor with the program branch it creates a new branch in

TABLE II: Branch Predictors included in the examples library.

Bimodal [18]
All versions of Two Level: GAg, GAs, PAs, SAp, etc. [31], [32]
GShare [14]
Generalized tournament [30]
2bc-gskew [33]
Hashed perceptron [34]
TAGE [15]
BATAGE [5]

which the outcome indicates the correct base predictor. On the other hand, the `track` function of the meta-predictor is always invoked with the program branch.

This example shows that, if the `train` function does the work of the `track` function as well, like in ChampSim and the CBP5 framework, it is not possible to write some types of meta-predictors without reimplementing the base predictors. It is not enough to have a function for conditional branches and another one for non-conditional branches, like the CBP5 framework does, because the metapredictor component trains for and tracks different branches.

Note as well how, on line 48, the meta-predictor includes a description of its components as part of its description. This is possible thanks to the flexibility of the JSON format.

## VII. EVALUATION

To evaluate the performance of MBPlib, we compared its running time against the CBP5 framework and ChampSim.

### A. Methodology

The branch predictors chosen for comparison were the ones included in the MBPlib examples library. In particular, we draw conclusions from the bimodal [18] and BATAGE [5] predictors. The first one is very simple, thus the vast majority of the running time is spent in the simulator code and it serves to measure the speed gained. The second one is one of the most complex state-of-the-art predictors. It includes multiple tables, has a prediction-overriding scheme based on priorities, has a non-trivial update policy and needs to generate random numbers. As such, it is computationally complex even among state-of-the-art predictors and we use it to measure how much speed gain we can expect for one of the slowest (in terms of simulation time) of the state-of-the-art predictors. In the case of ChampSim, we only ran the GShare and BATAGE predictors to show that simple and complex predictors have approximately the same running time.

To make the comparison fair, we used the same branch predictor implementations across the different simulators, with only small changes needed to comply with the different interfaces. For the same reason, all the executables were compiled with the same settings, using GCC with `-O3` and link time optimizations and targeting the underlying microarchitecture with `-march=native` and `-mtune=native`. ChampSim is configured with default parameters, similar to Intel's Ice Lake architecture [35], except for the branch predictor and the branch target buffer (BTB). We accompanied the GShare predictor with a 8 K-entry branch target buffer (BTB) and

a 4 K-entry GShare-like indirect target predictor [36], while for the BATAGE predictor, we used a 64 kB ITTAGE target predictor [37]. The rationale is that if we are going to simulate for performance, it makes sense to have a high-end target predictor accompanying a high-end branch predictor.

The executables generated by the CBP5 framework were run against the 223 training traces from the CBP5, which have from a few hundred million instructions for the shortest traces up to 55 billion for the longest trace. The executables generated by the ChampSim framework were run against the traces from the DPC3, a set of 95 traces generated from the SPEC17 suite [26]. Since the ChampSim framework is noticeably slower, we run only the first 100 million instructions from each trace.

### B. Performance Results

The results are shown in table III. The simulation time for bimodal (GShare in the case of ChampSim) measures the speedup gained inside the simulator code. On the other hand, the simulation time for BATAGE serves to give a measure of what final speedup we can expect in the slowest case. On average, we obtain, with respect to ChampSim, a  $923\times$  speedup for GShare and a  $134\times$  speedup for BATAGE. Note, however, that these speedups are so impressive because ChampSim is a cycle-accurate simulator. In fact, the fraction of time spent on branch predictor code in ChampSim is so small that the running time when using the GShare and BATAGE predictors is almost the same. With respect to the CBP5 framework, which is, like MBPlib, microarchitecture-agnostic, we obtain a  $18.4\times$  speedup in the simulator code (i.e., for Bimodal) which translates to a  $3.25\times$  worst-case scenario final speedup (for the BATAGE predictor).

An important fact about MBPlib is that, as is shown, the average running time for a simple predictor and 100 M instructions, which is already a good amount to measure the MPKI, is below 1 s. What this means is that the user can perform a couple of short and quick simulations with a set of 4 to 10 traces to reevaluate their design. This is very relevant when one is starting in the field and wants to prototype and test basic ideas fast. But it also allows using MBPlib during computer architecture classes.

### C. Simulation Results

Trace-based simulators always give the same results, provided that the user code is deterministic. In particular, MBPlib can be used as a replacement of the CBP5 framework, since it comes with the same set of traces. As part of the evaluation, we checked that the simulation results of both frameworks were identical.

The same is true for the ChampSim framework. However, since ChampSim sometimes simulates some extra instructions (due to the way it implements the instruction commit), there may be slight differences between the MPKIs calculated by ChampSim and MBPlib when the number of simulation instructions is very small.

TABLE III: Simulation time of MBPlib versus the CBP5 framework (CBP5 simulation traces) and ChampSim (100 million instructions).

	CBP5 Traces	CBP5	MBPlib	Speedup
<b>Bimodal</b>	Slowest	2.01 h	5.60 min	$21.49\times$
	Average	1.40 min	4.57 s	$18.38\times$
	Fastest	166.00 ms	148.41 ms	$1.12\times$
<b>Two-Level</b>	Slowest	2.05 h	5.92 min	$20.79\times$
	Average	1.47 min	5.00 s	$17.69\times$
	Fastest	26.00 ms	2.37 ms	$10.97\times$
<b>GShare</b>	Slowest	2.07 h	6.56 min	$18.90\times$
	Average	1.44 min	4.82 s	$17.88\times$
	Fastest	22.00 ms	4.86 ms	$4.53\times$
<b>Tournament</b>	Slowest	2.10 h	7.31 min	$17.20\times$
	Average	1.46 min	5.47 s	$15.96\times$
	Fastest	23.00 ms	6.12 ms	$3.76\times$
<b>2bc-gskew</b>	Slowest	1.77 h	7.90 min	$13.43\times$
	Average	1.36 min	6.73 s	$12.17\times$
	Fastest	39.00 ms	15.60 ms	$2.50\times$
<b>Hashed Perc.</b>	Slowest	1.92 h	17.12 min	$6.73\times$
	Average	1.48 min	14.38 s	$6.19\times$
	Fastest	42.00 ms	2.97 ms	$14.16\times$
<b>TAGE</b>	Slowest	2.18 h	32.54 min	$4.01\times$
	Average	1.73 min	28.01 s	$3.70\times$
	Fastest	14.00 ms	4.37 ms	$3.20\times$
<b>BATAGE</b>	Slowest	2.28 h	41.07 min	$3.34\times$
	Average	1.81 min	33.29 s	$3.25\times$
	Fastest	16.00 ms	4.84 ms	$3.31\times$

	100 M Instr.	ChampSim	MBPlib	Speedup
<b>GShare</b>	Slowest	1 h	1.7 s	$2167\times$
	Average	11.7 min	762 ms	$923\times$
	Fastest	3.7 min	3.7 ms	$55\,250\times$
<b>BATAGE</b>	Slowest	1 h	11.7 s	$309\times$
	Average	11.7 min	5.3 s	$134\times$
	Fastest	3.7 min	20 ms	$11\,050\times$

### D. Effects of the Compression Method on the Speedup

Since the SBBT traces were compressed with zstandard, which is a better compressor than gzip, it is important to measure which percentage of the speedup is gained thanks to using a new compression algorithm, rather than being thanks to the library implementation. Thus, we modified the CBP5 framework to allow it to read traces compressed with zstd and ran the same set of experiments in the modified version. For this, we recompressed the BT9 traces using the maximum compression level, as with the SBBT traces.

Table IV shows the speedup obtained by the CBP5 framework with the zstd traces. Since the speedup is only  $1.02\times$  to  $1.12\times$ , and MBPlib is  $3.25\times$  to  $18.4\times$  faster, we can conclude that the most significant part of the speedup is not thanks to the compression method. The speedup of MBPlib is due to other factors, such as the use of a stream-like format (SBBT), which avoids the cache misses of accessing a big hashed structure to read the branch metadata.



TABLE IV: Speedup of the CBP5 framework with Zstd.

(Averages)	CBP5 Gzip	CBP5 Zstd	Speedup
<b>Bimodal</b>	1.40 min	1.25 min	1.12 ×
<b>Two-Level</b>	1.47 min	1.32 min	1.12 ×
<b>GShare</b>	1.44 min	1.31 min	1.09 ×
<b>Tournament</b>	1.46 min	1.35 min	1.08 ×
<b>2bc-gskew</b>	1.36 min	1.32 min	1.03 ×
<b>Hashed Perc.</b>	1.48 min	1.44 min	1.03 ×
<b>TAGE</b>	1.73 min	1.69 min	1.02 ×
<b>BATAGE</b>	1.81 min	1.72 min	1.05 ×

### VIII. ADVANTAGES OF MBPLIB

ChampSim and the CBP5 framework have already found a place in the ecosystem of computer architecture simulation [13]. Nevertheless, MBPlib can stand as a good competitor and be a replacement for the researchers using the CBP5 framework, which is not maintained. In particular, MBPlib has the following advantages.

#### A. Modular

MBPlib is built with modularity in mind. First of all, it is distributed as three independent libraries to make it possible to use the utilities and examples library in a different simulator and not restrict your designs to MBPlib’s simulator. Secondly, it has modular output the user can complete. Thirdly, it defines a simple interface that makes it possible to use predictors as subcomponents. And lastly, it works as a library and lets the user depend on other tools to complement MBPlib.

#### B. Modern

To recent graduates, MBPlib will feel very modern. MBPlib uses a modern C++ standard, the de-facto C++ build system for cross-compilation, which is CMake, and a feature-complete JSON library. MBPlib codebase also resembles a modern library, which is something that helps attract collaborators to the project.

#### C. Fast

The utilities library makes the development faster and the codes shorter. The simulation library and the trace format speeds up the evaluation. Besides, MBPlib takes the everyday use cases into account and tries to handle them efficiently.

#### D. Lightweight

MBPlib’s traces use  $6.5 \times$  less disk space than the CBP5 traces. Only 769 MB of space are needed to store them.

#### E. Pedagogical

For educational purposes, it is better to use easy-to-use tools with simple installation. MBPlib satisfies these requirements and is fast enough to be used during a class. In addition, at the moment of writing, the examples library includes the predictors listed in table II. The most basic predictors, like bimodal and GShare, serve to introduce the topic. Tournament predictors and the 2bc-gskew predictor show more effective but still old examples. And the hashed perceptron, TAGE and BATAGE predictors are the state-of-the-art.

### IX. CONCLUSION

In this work we have presented the Modular Branch Prediction library, an open-source library for the simulation, development and teaching of branch predictors.

MBPlib arises as an alternative to the CBP5 framework, but it is significantly faster, more modular, modern and accessible. MBPlib runs  $18.4 \times$  faster than the CBP5 framework,  $923 \times$  faster if we measure it against ChampSim, and uses  $6.5 \times$  less space to store the same traces. In addition, MBPlib is built as a library instead of a framework. This means that the user code calls MBPlib and not the other way around. Thanks to that, it gives the user more freedom to use it in combination with other software. Moreover, MBPlib uses a modern build system, produces a more detailed output which can include user data and favors designs written with composability in mind. Besides, MBPlib offers a utilities library, which eliminates the need to reimplement common logic, and an examples library, which is one of the largest collections of branch predictor implementations. Thus, we reckon MBPlib can achieve a new lower bound to enter the field and has the potential to help current and new researchers, and we hope to see it getting adopted by the community.

### ACKNOWLEDGEMENTS

We would like to thank Juan M. Cebrian for his review of this paper and his helpful comments and Daniel A. Jiménez for sharing with us the traces of the CBP5 competition, which are now unavailable online.

This work has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 819134) and the Grant TED2021-130233B-C33 funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR.

### REFERENCES

- [1] “Popping the hood on golden cove”, Chips and Cheese. (Dec. 2021), [Online]. Available: <https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove>.
- [2] *Graviton 3: First impressions*, <https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions>, May 2022.
- [3] S. Mittal, “A survey of techniques for dynamic branch prediction”, *Concurrency and Computation Practice and Experience*, vol. 31, 2018. DOI: 10.1002/cpe.4666.
- [4] A. Seznec, J. S. Miguel, and J. Albericio, “The inner most loop iteration counter: A new dimension in branch history”, in *48th Int’l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 347–357.
- [5] P. Michaud, “An alternative tage-like conditional branch predictor”, *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 3, 30:1–30:23, Apr. 2018.

- [6] S. Zangeneh, S. Pruetz, S. Lym, and Y. N. Patt, “Branchnet: A convolutional neural network to predict hard-to-predict branches”, in *53rd Int’l Symp. on Microarchitecture (MICRO)*, Oct. 2020, pp. 118–130.
- [7] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, “Whisper: Profile-guided branch misprediction elimination for data center applications”, in *55th Int’l Symp. on Microarchitecture (MICRO)*, Oct. 2022, pp. 19–34.
- [8] *The 1st jilp championship branch prediction competition (cbp-1)*, 2004. [Online]. Available: <https://jilp.org/cbp/>.
- [9] *2nd championship branch prediction (cbp-2)*, 2006. [Online]. Available: [https://camino.rutgers.edu/cbp2/%20\(down\)](https://camino.rutgers.edu/cbp2/%20(down)).
- [10] *The 3rd championship branch prediction (cbp-3)*, 2001. [Online]. Available: <https://jilp.org/jwac-2/>.
- [11] *4th championship branch prediction competition (cbp-4)*, 2014. [Online]. Available: <https://jilp.org/cbp2014/>.
- [12] *4th championship branch prediction competition (cbp-5)*, 2016. [Online]. Available: <https://jilp.org/cbp2016/>.
- [13] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, “The championship simulator: Architectural simulation for education and competition”, *CoRR*, vol. abs/2210.14324, 2022.
- [14] S. McFarling, “Combining branch predictors”, 1993.
- [15] A. Seznec and P. Michaud, “A case for (partially) tagged geometric history length branch prediction”, *Journal of Instruction-level Parallelism - JILP*, vol. 8, Feb. 2006.
- [16] M. Breughe, *Maximizing branch behavior coverage for a limited simulation budget*, Samsung Austin R&D Center, Jun. 2016. [Online]. Available: [http://www.jilp.org/cbp2016/slides/mbreughe\\_WorkloadSelection.pptx](http://www.jilp.org/cbp2016/slides/mbreughe_WorkloadSelection.pptx).
- [17] *The 3rd data prefetching championship (dpc-3)*, Jun. 2019. [Online]. Available: <https://dpc3.compas.cs.stonybrook.edu/>.
- [18] J. K. F. Lee and A. J. Smith, “Analysis of branch prediction strategies and branch target buffer design”, EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-83-121, Aug. 1983. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1983/6335.html>.
- [19] *Smaller and faster data compression with zstandard*, Facebook, 2016. [Online]. Available: <https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/>.
- [20] *Gnu gzip*, Free Software Foundation. [Online]. Available: <https://www.gnu.org/software/gzip/>.
- [21] A. M. Devices, “Amd x86-64 architecture programmer’s manual volume 2: System programming”, Tech. Rep., 2022.
- [22] Intel, “Intel® 64 and ia-32 architectures software developer’s manual”, Tech. Rep., 2022.
- [23] Arm, “Arm® architecture reference manual for a-profile architecture”, Tech. Rep., 2015.
- [24] D. A. P. John L. Hennessy, *Computer Architecture: A Quantitative Approach, 4th Edition*, 4th ed. Morgan Kaufmann, 2006, ISBN: 0123704901; 9780123704900.
- [25] *Spec cpu 2017 traces for champssim*, <https://hpca23.cse.tamu.edu/champssim-traces/speccpu/index.html>, Feb. 2019.
- [26] Standard Performance Evaluation Corporation, *SPEC CPU2017*, 2017. [Online]. Available: <http://www.spec.org/cpu2017>.
- [27] *Champsim microarchitecture simulator*. [Online]. Available: <http://github.com/ChampSim/ChampSim> (visited on 2023).
- [28] Intel, *Pin - a dynamic binary instrumentation tool*. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html> (visited on 2022).
- [29] *Json for modern c++*, <https://github.com/nlohmann/json>. [Online]. Available: <https://json.nlohmann.me/>.
- [30] M. Evers, T.-Y. Yeh, and Y. N. Patt, “Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches”, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996. DOI: 10.1145/232973.232975.
- [31] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction”, *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA ’92)*, 1992. DOI: 10.1145/123465.123475.
- [32] —, “Alternative implementations of two-level adaptive branch prediction”, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, 1992. DOI: 10.1109/ISCA.1992.753310.
- [33] A. Seznec and P. Michaud, “De-aliased hybrid branch predictors”, Inria, Tech. Rep., 1999.
- [34] D. Tarjan and K. Skadron, “Merging path and gshare indexing in perceptron branch prediction”, *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, pp. 280–300, Sep. 2005, ISSN: 1544-3566. DOI: 10.1145/1089008.1089011. [Online]. Available: <https://doi.org/10.1145/1089008.1089011>.
- [35] I. E. Papazian, “New 3rd gen Intel® Xeon® Scalable processor (Codename: Ice Lake-SP)”, in *32nd HotChips Symposium*, Aug. 2020, pp. 1–22.
- [36] P.-Y. Chang, E. Hao, and Y. N. Patt, “Target prediction for indirect jumps”, in *24th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1997, pp. 274–283.
- [37] A. Seznec, “A 64-Kbytes ITTAGE indirect branch predictor”, in *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, 2011.