Rebasing Microarchitectural Research with Industry Traces

Josué Feliu[1] Arthur Perais[2] Daniel A. Jiménez[3] Alberto Ros[4]

[1]*Universitat Politècnica de València, València, Spain*
[2]*Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, Grenoble, France*
[3]*Texas A&M University, College Station, USA*
[4]*University of Murcia, Murcia, Spain*

Abstract

Microarchitecture research relies on performance models with various degrees of accuracy and speed. In the past few years, one such model, ChampSim, has started to gain significant traction by coupling ease of use with a reasonable level of detail and simulation speed. At the same time, datacenter class workloads, which are not trivial to set up and benchmark, have become easier to study via the release of hundreds of industry traces following the first Championship Value Prediction (CVP-1) in 2018. A tool was quickly created to port the CVP-1 traces to the ChampSim format, which, as a result, have been used in many recent works. In this paper, we revisit this conversion tool and find that several key aspects of the CVP-1 traces are not preserved by the conversion. We therefore propose an improved converter that addresses most conversion issues as well as patches known limitations of the CVP-1 traces themselves. We evaluate the impact of our changes on two commits of ChampSim, with one used for the first Instruction Championship Prefetching (IPC-1) in 2020. We find that the performance variation stemming from higher accuracy conversion is significant.

1. Introduction

Any computer engineer or scientist is familiar with the idiom *garbage in, garbage out*. Contextualized for the computer architecture research community, this means that the evaluation of an idea is only as good as the workloads used in the evaluation. The definition of *good* is already ambiguous, as it can entail *relevant* (to users, industry, academia) or *representative* (of different enough classes of algorithms or behaviors), or both. Once good workloads have been chosen, they are usually simulated via emulation, e.g., QEMU [1], gem5 [2], or by extracting traces and feeding those traces to the simulator, e.g., ChampSim [3]. In both approaches, some information may be lost in translation, such that a good workload is incorrectly simulated because either instruction semantics are not correctly implemented, or the information contained in the trace is intrinsically incorrect, or incorrectly processed.

Crafting and bringing up good workloads is an arduous task, as exemplified by the fact that most microarchitectural work remains focused on the SPEC CPU benchmarks [4, 5]. Thus, the release of good workloads e.g., by industry, is a boon for the research community, as it provides relevant and –a priori– representative workloads that require a minimal amount of work to bring up.

Over the past two decades, many microarchitecture championship workshops took place at flagship conferences to encourage research in branch prediction [6–10], cache replacement policies [11, 12], data prefetching [13–15], instruction prefetching [16], value prediction [17] and memory scheduling [18]. While each championship generally used its own infrastructure, several of the more recent ones elected to rely on the ChampSim [3] trace-based simulator [12, 15, 16, 19]. ChampSim began as an internal effort at Intel before being taken over by the current maintainers at Texas A&M University. Intel has been involved in many of the recent microarchitectural competitions, so it was natural for ChampSim to be used. Indeed, the name "ChampSim" is an abbreviation for "Championship Simulator." The convergence to ChampSim is especially interesting to the community as it provides a reasonably complex simulator that models a non-trivial pipeline and features state-of-the-art cache replacement policies and data/instruction prefetchers, and is therefore quite convenient for the first order study of those "bolt-on" microarchitectural structures. Case in point, several papers published in top-tier conferences have used ChampSim as their experimental workbench [20–29].

Nevertheless, several championships were organized –mostly– by industry members of the research community, e.g., Intel [8, 9, 11, 14–16], Samsung [10], and Qualcomm [17]. This aspect is of particular interest as, in some cases, organizers provided an infrastructure and especially program traces generated from non-standard workloads [6, 9–11, 17].

The traces used for the Fifth Branch Prediction Championship [10] are also from industry: they were donated by Samsung as a distilled subset of the traces used in the design of the Samsung Exynos M-series of SoCs [30]. These traces are not quite as useful as the traces Qualcomm released after the first Championship Value prediction (CVP-1) [17], since they focus on mobile workloads and only contain information about branch instructions, not other instructions. Thus, they can be used to measure statistics related to front-end performance such as L1i cache misses, branch mispredictions, and instruction address translation

*Institute of Engineering Univ. Grenoble Alpes

behavior, but they cannot be used to measure instructions-per-cycle, data cache misses, or instruction-level parallelism.

Conversely, the CVP-1 "public" [31] and "secret" [32] Aarch64 traces generated at Qualcomm are of large interest to the research community as i) They are numerous (135 and 2013 traces respectively) ii) They cover a wide range of workloads of interest to industry: Compute INT/FP, cryptography and datacenter/server iii) They embed output register values, allowing studies that rely on actual program values and iv) They include system activity, which is typically not the case for traces obtained with Pin [33]. Unfortunately, those traces are anonymized, meaning that the actual workload is not known, and some information has been removed from the traces (e.g., Execution Level and Address Space IDentifier, instruction bytes, exact opcode, addressing mode, etc.).

Given those two observations, being able to run CVP-1 traces in ChampSim would provide the research community with a large amount of good traces running on an easy to use infrastructure. In fact, a trace converter, *cvp2champsim*, is already available in the ChampSim repository, and a subset of the CVP-1 traces were converted for the first Instruction Prefetching Championship (IPC-1) [16], which used ChampSim as infrastructure. However, ChampSim was initially designed to read x86 traces, and, as our findings show, the conversion phase from Aarch64 is non-trivial.

Moreover, the CVP-1 traces README describes a handful of limitations that can lead to inaccuracies, although most can in fact be patched [31, 32]. This is of particular interest as a careful understanding of traces is fundamental to model their execution properly. For example, such an issue was patched in the –cancelled– CVP-2 [34] simulator to improve simulation fidelity over CVP-1. In a nutshell, the trace format does not allow attaching a latency or an operation type to an output register: The information is attached to the instruction. This is problematic for loads that use Pre/Post Indexing Increment addressing mode, in which the base register used for address calculation is also updated by the instruction. A first issue is that in the CVP-1 simulator, the total access size of the instruction is computed as the transfer size times the number of output registers. However, since one of the output is not populated from memory, the total access size is actually incorrect, leading to an incorrectly modeled data memory footprint. Second, updating the base register requires a simple addition, such that the updated base register should be made available to dependents immediately after address calculation. However, since it is treated as being populated from memory, it becomes available to dependents when data comes back from the memory system. This hinders memory level parallelism as any instruction depending on the base register may, in the worst case, have to wait for a DRAM access to compute its address. This limitation was made known to the ChampSim community in January 2021,[1] but is currently not addressed in the existing *cvp2champsim* converter.

1. https://github.com/ChampSim/ChampSim/issues/112

In that spirit, this paper revisits the CVP-1 traces and performs a thorough analysis allowing a better conversion of the CVP-1 traces to the ChampSim format. During this analysis, we discovered some design decisions in the initial conversion to the IPC-1 format affecting performance and even the final outcome of the IPC-1. One example is that a very high misprediction rate was observed for *return* instructions in 10 IPC-1 workloads out of 50, despite ChampSim implementing a Return Address Stack (RAS) [35]. These additional mispredictions were in fact caused by incorrectly converting the CVP-1 traces. Other examples are register dependencies that do not survive the conversion in specific cases and the absence of load/store pairs handling.

Using the latest *develop* version of ChampSim (see Section 4 for the experimental framework details) and comparing the traces converted with the original *cvp2champsim* converter and our improved converter, we find that the IPC differs by more than 5% in 51 out of the 135 CVP-1 public traces. This illustrates the importance of analyzing and understanding freely available traces to accurately convey the characteristics of the original applications to a particular simulation infrastructure.

2. Design Decisions in the Original *cvp2champsim*

The original *cvp2champsim* converter program was written out of a requirement of being able to model workloads with interesting front-end behavior in ChampSim. The intent was to study instruction cache and branch predictor optimizations, for which the Aarch64 to x86 conversion is comparatively straightforward. Some of the CVP-1 workloads had been identified as having very large instruction working sets, the kind of which were as yet unavailable in other open benchmarks. The original emphasis of the converter was therefore to generate ChampSim traces that captured this particular front-end behavior.

Specifically, ChampSim deduces branch type using x86-specific registers, while CVP-1 encodes Aarch64 instructions, so great care was taken to make converted instructions appear to follow the x86 semantics for branches, as we will detail in the next Section. Unfortunately, this did not prevent the introduction of a bug that causes some call and return pairs not to align properly, leading to an inordinate number of return target mispredictions in ChampSim.

Conversely, the converter put less emphasis on correct conversion of aspects of execution unrelated to the front-end. For example, the converter fails to properly handle instructions that have more than one destination register. Moreover, the CVP-1 traces themselves are known to be inaccurate with respect to certain kinds of dependencies due to an ambiguity in the tracing methodology. This flaw does not affect front-end studies (e.g. branch prediction, icache replacement, icache prefetching, iTLB management, and BTB management) but is important for data-intensive workloads. The CVP-1 traces come with an explicit warning about this limitation [31, 32].

TABLE 1: Summary of the proposed trace conversion improvements.

Instruction type	Improvement name	Modifications to the converter
Memory	mem-regs	Convey all dependencies between the registers written by memory instructions and the instructions that read from them.
	base-update	Make base registers available after the latency of an ALU instruction rather than after the latency of the memory access.
	mem-footprint	Access all cachelines accessed by the instruction.
Branch	call-stack	Fix the identification of returns.
	branch-regs	Convey all dependencies between the registers read by branch instructions and the instructions that generate them.
	flag-reg	Add the flag register as dethe stination of ALU and FP instructions that do not have any destination register so that branches reading from flags depend on them.

The converter was used to produce the traces used in the IPC-1 championship [16], resulting in traces that correctly modeled front-end behavior (modulo return misses) but were inaccurate with respect to workloads with a high sensitivity to instruction-level parallelism. Unfortunately, the deficiencies in the converter were not addressed and researchers began using it for more general-purpose microarchitectural studies, which compounds with the intrinsic limitations of the CVP-1 traces.

3. Analysis of Traces and Translation Decisions

ChampSim traces have a fixed format, with each instruction occupying 64 bytes. These bytes are organized in: i) instruction pointer (8B), ii) "is branch" (1B), iii) branch taken (1B), iv) destination registers (2×1B), v) source registers (4×1B), vi) memory destinations (2×8B), and vii) memory sources (4×8B). The trace format is strict: an arithmetic instruction will still occupy the 64 bytes even if it does not have any memory source or destination. Fortunately, ChampSim traces tend to be highly compressible so this format does not necessarily require exorbitant storage.

It is worth noticing that the ChampSim trace format does not include a field to indicate the operation type. Only a flag (the `is branch` field takes a byte but it is used as a boolean) indicates if the instruction is a branch or not. To determine if the instruction is a load or a store, ChampSim looks, respectively, at the memory sources and destinations. Instructions that are not branches, loads, or stores are considered arithmetic instructions. Furthermore, the trace format does not encode the branch type. ChampSim resolves the type of the branch based on the registers from which the branch instructions read from or write to. For example, if an x86 instruction reads and writes both the stack pointer and instruction pointer, but does not read other registers, it must be a direct call instruction.

Next, we discuss different issues we have identified in the original *cvp2champsim* converter and describe the improvements we propose to make the ChampSim traces converted from the CVP-1 traces more accurate. Table 1 summarizes these improvements.

3.1. Memory Instructions

The Aarch64 ISA, as any other ISA, implements memory operations with multiple addressing modes. Two examples are `LDR X1, [X0, #12]!`, a load with pre-indexing increment addressing that adds the immediate to X0 and uses the result as the address to load 8 bytes into X1, and `LDP X1, X0, [X0]`, a load pair that uses addresses X0 and X0+8 to load 8 bytes into X1 and the next 8 bytes into X0. The CVP-1 traces, however, do not encode the addressing mode. They only contain the memory address, access size for one register, and source and destination registers for the instructions. This limitation makes the two previous instructions seemingly indistinguishable: both are loads with one source register (X0) and two destination registers (X1 and X0). Consequently, the original *cvp2champsim* converter handles both instructions in the same way: It generates a load with two source registers (X0 and X1), one destination register (X1), and one source memory address (the one stored in the CVP-1 instruction).

This conversion therefore makes three approximations. First, it only includes one destination register when both instructions write two. Second, there is no distinction between the memory-loaded register and the base register. Third, they use a single address as a memory source, independently of how many memory accesses are performed and what is their size. This will typically misrepresent accesses that cross cachelines.

3.1.1. Improvement *mem-regs*: Keep All Destination Registers (and Only Them) From the CVP-1 Traces.

The CVP-1 traces comprise instructions with a number of destination registers ranging from zero to three. Prefetch loads and some stores have no destination register in the CVP-1 traces. Stores that perform base update or that are store exclusive have a destination register. In addition, because the CVP-1 traces only include the general purpose registers as source and/or destination registers, many arithmetic instructions that modify the flag register appear in the CVP-1 trace with no destination register at all. They will be discussed in Section 3.2.3. Instructions with multiple destination registers are loads that either update the base register (e.g., the LDR in the previous example), are load pairs (e.g., the LDP in the previous example), or

are vector loads (e.g., LD2, LD3, and LD4) and may involve multiple memory accesses.

Despite this variety in the number of destination registers across different instructions, the original *cvp2champsim* converter forces all instructions except branches to have a single destination register. In the case of prefetch loads and stores that do not have any destination register, register X0 is added. This creates dependencies between these instructions and younger instructions that read from register X0 that do not exist in the original CVP-1 trace. Similarly, for loads with multiple destination registers, only the first one is preserved. As a result, dependencies between these load instructions and younger instructions that read from the missing destination registers are missing from the converted traces.

To address these issues, our improved *cvp2champsim* converter does not add any destination register for prefetch loads and stores with no destination register and preserves all the destination registers specified in the CVP-1 traces.[2] For example, the two sample loads introduced earlier now feature both X0 and X1 as destination registers, while they only featured X1 with the original converter.

3.1.2. Improvement *base-update*: Dependencies Resolved with Different Latency for Destination Registers of Memory Instructions.

Taking the previous example, after fixing the number of destination registers, our converter would generate a load with one source (X0) and two destination registers (X1 and X0) for both the LDR and LDP. That is, in both cases, registers X0 and X1 would only be available for dependent instructions after the load completion. From the point of view of the trace, this is accurate for the load pair, as both registers should only be available after the corresponding memory accesses complete, whatever their latency. However, this is not accurate for the pre- and post-indexing increment LDR as, in that case, X0 should be available to dependent instructions after the latency of the arithmetic operation that updates the base register X0.

To make the converted traces more accurate regarding memory operations that update the base register, we attempt to infer their addressing mode and determine if a particular instruction performs a base update. To this end, we leverage the heuristic proposed by the trace maintainer [36] with minor improvements. The heuristic infers the addressing mode based on the source and destination registers, the output value for the destination registers present in the CVP-1 trace, and the current value of the registers kept in a data structure in the trace reader and updated with the value written to the destination registers by the trace instructions. In short, this inference is done by checking the number of source and destination registers, whether any of the source registers is also a destination, and whether the effective address difference compared to the value written to the candidate base register fits within an immediate offset. Note that this inference is best effort as it is not always possible to get the exact answer.

Once the addressing mode has been obtained, we further differentiate between pre- and post-indexing increment to determine if the arithmetic operation that updates the base register should come before the memory instruction, or after. To this end, we only need to compare the effective address of the memory instruction with the value written to the base register: if they match, the base register is updated before the memory access is performed; otherwise, the base register is updated after the memory instruction takes place. Luckily, the CVP-1 trace instructions include the value written to the destination registers, making this comparison straightforward.

Finally, the converter writes the two instructions to the trace file. We assign the PC of the CVP-1 trace instruction to the first instruction (i.e., the base register update ALU in case or pre-indexing increment or the memory instruction in case of post-indexing increment) and assign PC + 2 to the second instruction (i.e., the memory instruction in case of pre-indexing increment and the base register update ALU in case of post-indexing increment). The only inaccuracy this conversion could cause is that it encourages the simulator to handle the instructions as two micro-ops, occupying two entries in the ROB and IQ and requiring double the bandwidth across the pipeline even though a particular microarchitecture could implement that behavior within a single micro-op. This is not a fundamental issue as the simulator logic could identify this and bundle the two trace instructions together.

The importance of this improvement lies in making base register available to dependent instructions after the latency of the ALU operation. Notice that without this improvement, in the case of a load that performs base update and results in a main memory access, the base register would only become available after the memory access is completed, delaying the execution of younger instruction that depend on the base register. Correctly identifying pre- and post-indexing increment loads also enables to better convey the applications' data memory footprint, which is the improvement we describe next.

3.1.3. Improvement *mem-footprint*: Improving the Memory Footprint.

Finally, we look at the access size of the memory instructions. Inherent limitations of the CVP-1 and ChampSim trace formats make it hard to correctly convey the memory footprint of the original applications to ChampSim. First, the CVP-1 trace format merges, within the destination registers of load instructions, registers populated from memory with address registers updated before or after the memory access. Furthermore, the Aarch64 ISA, hence the CVP-1 traces, include load pairs and vector loads that may result in two 64B cacheline accesses. Consequently, computing the total

2. A handful instructions include more than four source registers (e.g., *compare and swap pair*). However, ChampSim only allows four of them by default. Because these instructions are very infrequent and we aim to limit our changes to ChampSim, our converter only conveys the first four source registers to the ChampSim traces in these cases.

access size of the instructions as the transfer size –which is what the CVP-1 trace contains– times the number of output registers would overestimate the total memory access size. Even worse, ChampSim does not model the access size of memory operations and so ChampSim traces do not include that information altogether.

Our solution to improve on this and better convey the data memory footprint of the original applications to ChampSim is to make the converter determine the memory transfer size of instructions and whether this transfer size involves single or multiple 64B cachelines. To this end, we first identify the memory addressing mode and determine if the instruction updates the base register or not, which is an extension of the logic added for improvement *base-update* (Section 3.1.2). This allows us to correctly calculate the transfer size of the instructions. After that, if a memory operation spans two cachelines, we add the address of the second cacheline as a second memory source or destination in the converted instruction, depending on whether it is a load or a store.

In addition to that, our converter also identifies 64B stores as `DC ZVA` instructions, which zeroes a naturally aligned block of 64 bytes. Even though we did not find any case across the CVP-1 public traces, the instruction is architecturally allowed to have a non-aligned address. Thus, since `DC ZVA` instructions, by definition, touch a single cacheline, our converter always aligns their effective address to a cacheline boundary.

3.2. Branch Instructions

ChampSim models a relatively detailed processor front-end, including a branch target buffer (BTB). Modern BTBs usually store the branch type in the main BTB table and implement separate structures to predict the targets of indirect branches and returns more accurately. ChampSim differentiates among different types of branches even though the ChampSim traces do not encode the branch type directly. ChampSim resolves the type of branches based on the registers from which the branch instructions read from or write to. For example, a conditional branch reads from and writes to the instruction pointer register, reads the flag register, and does not read nor write the stack pointer register.

The CVP-1 traces only distinguish between three different types of branches (conditional, unconditional direct, and unconditional indirect), ChampSim differentiates among six types. To further refine the branch type provided in the CVP-1 traces, the *cvp2champsim* converter checks the instruction class and whether it reads from and writes to register X30[3] or not. In the original converter, a branch instruction reading from register X30 is classified as a return. Otherwise, the branch is either a call, if it writes to register X30, or a jump, if it does not. Calls and jumps are further classified as direct or indirect depending on the instruction class.

3. The link register in the Aarch64 ISA

This conversion first misclassifies some calls as returns. Second, it does not convey the original source registers of branches in the CVP-1 traces to the converted traces. Third, it ignores that some arithmetic instructions that modify the flag register do not report that register among their destination registers.

3.2.1. Improvement *call-stack*: Fixing the Call Stack.

The branch type identification of the original *cvp2champsim* is based on the use that branches make of register X30 and on the type of branch encoded within the CVP-1 trace instruction. The converter correctly identifies the types of the branches in most cases but fails to correctly classify branches that read and write to register X30. These branches are classified as returns when they are actually calls. This misclassification could go unnoticed in many traces that have no indirect calls reading from register X30 (or a negligible number of them). However, in other traces such as *srv_3* or *srv_62*, this misclassification results in a relatively high target misprediction rate for returns. This was unexpected as a large enough Return Address Stack is expected to provide close to perfect return target prediction.

Our improved *cvp2champsim* converter makes sure that only unconditional branches that read from register X30 and do not write to any register are identified as returns. Otherwise, if they write to register X30, they are classified as calls, either direct or indirect depending on the instruction type in the CVP-1 trace. Unconditional branches that are neither calls nor returns are classified as direct or indirect jumps, as done in the original *cvp2champsim* converter.

3.2.2. Improvement *branch-regs*: Keeping the Original Dependencies of Branches with Older Instructions.

When converting a branch instruction from the CVP-1 trace, the original *cvp2champsim* sets the instruction pointer, stack pointer, and flag registers as source and/or destination registers adequately so that ChampSim correctly identifies the branch type. In this process, however, the original source registers present in the CVP-1 trace are not preserved. This affects some conditional branches (`tb(n)z, cb(n)z`), indirect calls, and indirect jumps, which have general purpose source registers in the CVP-1 trace. Consequently, the dependency between these branch instructions and producer instructions in the CVP-1 trace is lost. This issue might not impact performance significantly in many cases as it is likely that, depending on a previous instruction or not, the branch can still be resolved in the shadow of the execution of older instructions. However, there is a particular case in which the impact on performance can be high: A branch that depends on a long-latency load. In this case, the situation is completely different depending on whether the load-to-branch dependency is preserved or not. If the dependency is kept, a branch misprediction could only be corrected after the long latency load is resolved, exposing the branch misprediction penalty on the critical path. Conversely, if the dependency is not kept, the branch

misprediction penalty could be hidden in the shadow of the memory access. Furthermore, the negative impact on performance can be aggravated with a decoupled front-end and frequent BTB misses, as the prefetching effect of the decoupled front-end can be drastically reduced.

Our improved *cvp2champsim* converter preserves all the register dependencies among instructions present in the CVP-1 trace. To this end, in the case of branches, it adds the source registers present in the CVP-1 trace instruction to the required (depending on the branch type) instruction pointer, stack pointer, and flag registers.

More precisely, we make the following changes. For indirect jumps and calls, we add the source register of the CVP-1 trace instructions as source register of the converted instruction and avoid adding register X56, which was added by the original *cvp2champsim* to convey the *reads other register* information to ChampSim so that it can identify these branch types correctly. Note that these branches did not depend on the older instruction generating register X56 in the original trace.

For conditional branches, we distinguish two cases. If the CVP-1 trace instruction has a source register, we add the source register to the converted instruction and avoid adding the flag register as the original *cvp2champsim* does. These branches are likely `cb(n)z` and `tb(n)z`, which jump based on the content of a general-purpose register. Otherwise, if the CVP-1 trace instruction does not have any source register, we keep adding the flag register, as done in the original converter. These branches should jump based on the content of the flag register but, because the CVP-1 traces do not include special-purpose registers, such information is missing. This missing dependency will be addressed in the next section.

To convey the register dependence between a conditional branch and the instructions that generate its source registers, we opted for allowing conditional branches to depend on any register. However, this does not align with x86 semantics for branches, as conditional branches always depend on the flag register. An alternative implementation would be to split these conditional branches into two instructions: an arithmetic instruction taking the source registers of the branch and writing the flag register, and a conditional branch that reads from the flag register. This resembles our solution for the *base-update* improvements. Nevertheless, in this case, we consider that increasing the instruction count and resource occupancy by splitting these conditional branches into two instructions brings more inaccuracy than working around x86 semantics for branches.

Replacing the flag register with the source register in the trace instruction for some conditional branches, conflicts with the way ChampSim identifies branches. ChampSim deduces that a branch is conditional when it reads and writes the instruction pointer, does not write the stack pointer, reads flags, and does not read anything else. The only way we can address this shortcoming is by patching ChampSim to change the current condition of *reading flags and not reading any other register* to identify a conditional branch, to *reading either flags or other registers*.

In addition, we also need to modify the way ChampSim deduces indirect jumps. ChampSim checks if a branch is an indirect jumps before checking if it is a conditional branch, and a branch is deduced as an indirect jump if it writes the instruction pointer, does not read the stack pointer, does not read flags, and reads anything else. Based on these conditions, the *new* conditional branches that do not read from the flag register but read from other registers would be misclassified as indirect jumps. To address this shortcoming, we propose adding the *does not read the instruction pointer* check to the conditions that a branch should meet to be identified as an indirect jump by ChampSim. The change should be safe, as x86 indirect branches are always absolute and thus should not read the instruction pointer. Finally, adding the source register from the CVP-1 trace to indirect calls and jumps does not affected their identification as they already read from other registers.

A known limitation of this improvement is that we cannot keep register X30 as a destination register of calls since they should write to the instruction pointer and stack pointer registers in order for ChampSim to identify them correctly and, by default, the maximum number of destination registers in ChampSim is two. Seeking to limit our changes to ChampSim, we decided to not increase the number of destination registers and miss the dependency between calls and the following consumers that read X30 (until a non-branch instruction writes it). This dependency only affects, on average, 0.87% of the instructions across the CVP-1 public traces and never affects the (possible) dependency between calls and returns, which is preserved through the stack pointer register (written by calls and read by returns).

3.2.3. Improvement *flag-reg*: Adding the Missing Flag Register as Destination of ALU and FP Instructions with No Destination Registers.

In both Aarch64 and x86, the flag register is a special-purpose register that contains information about the status of arithmetic and logic operations. Among others, it indicates if the result of an operation is negative, zero, or if it resulted in an overflow. Consequently, it is widely used by conditional branches, which can determine whether a branch should be taken or not depending on the flag register, updated by a previous arithmetic or logical operation. Despite its importance, the CVP-1 traces only include the general purpose registers as source and destination registers of the instructions, and therefore many dependencies between branches and previous instructions are missing in the CVP-1 traces themselves.

Because ChampSim assumes that conditional branches always read from the flag register, the original *cvp2champsim* converter sets ChampSim's flag register as the source register for conditional branches. This does not solve the problem of the missing dependencies of conditional branches in the CVP-1 trace as no instruction writes to the flag register. To properly establish the dependencies between arithmetic and logical operations, and conditional

branches, we add the flag register as the destination register of all ALU instructions that have no destination registers, as recommended by the CVP-1 traces' README [31, 32]. This is slightly pessimistic as syscall instructions will be marked as generating the flags, but those are relatively rare and in any event should cause a pipeline flush so having them generate flags would not be likely to affect dynamic scheduling. In addition, and although they are significantly less frequent, we add the flag register to (arithmetic) FP instructions without a destination register. With this change, we recover the dependency of conditional branches from previous instructions through the flag register that is not present in the CVP-1 trace.

4. Methodology and Results

After translating the CVP-1 traces to ChampSim format with our enhanced converter, we evaluate the impact of the modifications using the current version of the *main* branch of ChampSim [37] (commit *2bba2bd*). Our changes on the front-end include a 16K-entry BTB and 64KB ITTAGE [38] and TAGE-SC-L [39] predictors. We model an ip-stride prefetcher at the L1D cache and a next-line prefetcher at the L2 cache, in an attempt to mimic Intel's Icelake prefetching mechanism [40].

We focus first on the impact of the modifications on public CVP-1 traces [31] performance, although the secret CVP-1 traces [32] can also be translated and characterized using our enhanced *cvp2champsim* trace converter. We run the traces until the end without warm-up (Section 4.1). Using the same methodology, we then analyse the impact of the most significant trace improvements (Section 4.2).

Through personal communication with the organizers of IPC-1 we were able to match the 50 IPC-1 traces back to the original CVP-1 secret traces. With that information we first perform a characterization of the IPC-1 traces (4.3). Then we evaluate the eight prefetchers accepted at IPC-1, using the same traces as in the championship but after our fixes. For this study we employ the ChampSim version provided in IPC-1, and we warm up for 50 million instructions and report statistics for 50 million instructions more, as was done in the contest. (Section 4.4)

4.1. Impact on Projected Performance

Figure 1 shows the IPC variation of the geometric mean of IPC for the proposed improvements compared to the original traces across the CVP-1 public traces. Starting with the memory improvements, we observe that making the base register in memory operations available after the latency of an ALU operation (imp. *base-update*) allows dependent instructions to execute earlier. This accelerates the back-end and improves the IPC, by 1.9%, on average. Improving the memory footprint of the converted traces by adding a second memory address to memory instructions that involve two cachelines (imp. *mem-footprint*) has a very small negative impact on performance (–0.04%) because most memory operations access a single cacheline. Finally, strictly keeping the original destination registers of loads

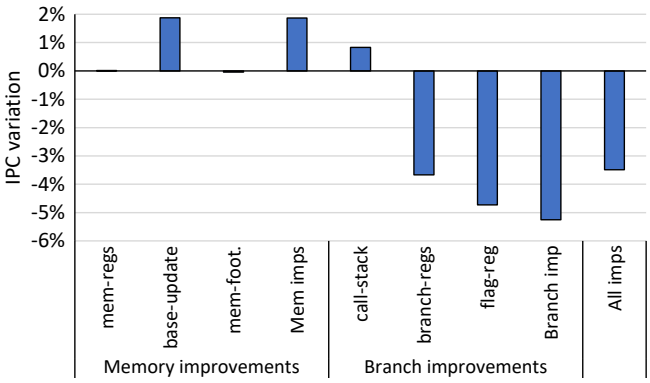

Figure 1: IPC variation of the geometric mean IPC across the CVP-1 public traces when applying the different improvements compared to the original *cvp2champsim* converter.

(imp. *mem-regs*) also has a negligible impact on IPC (0.01%). Overall, the memory improvements altogether increase the IPC by 1.9% on average.

Regarding the branch improvements, we observe that adding the flag register to ALU and FP instructions without any destination register (imp. *flag-reg*) and keeping the original source and destination registers of branches in the CVP-1 trace (imp. *branch-regs*) have a significant negative impact on performance. On average, improvements *flag-reg* and *branch-regs* reduce IPC, respectively by 4.7% and 3.7%. On the contrary, fixing the call stack (imp. *call-stack*) improves performance, on average, by 0.8%. As we will discuss later, the performance impact of this improvement is higher but only affects a subset of the traces. The three branch improvements altogether reduce IPC by 5.3% on average. When putting all memory and branch fixes together, IPC reduces by 3.5% on average across the CVP-1 public traces.

It is worth noting that the performance impact that improvements *branch-regs* and *flag-reg* have in isolation overlaps when they are applied together. Improvement *flag-reg* adds the flag register as a destination register for all ALU and FP instructions without one. Thus, it makes all conditional branches depend on an older ALU or FP instruction. However, when applying *branch-regs*, in the case of conditional branches that have a source register in the CVP-1 instructions, it replaces the flag register with such a register, thus reducing the impact that *flag-reg* has in isolation.

The average provides a general picture of how the different improvements affect performance but does not clearly reflect their impact, as the improvements affect the distinct traces in an irregular way. Figure 2 shows the IPC variation caused by the different improvements when applied individually and all together to each CVP-1 public trace. Notice that the traces are sorted from highest IPC increase to highest IPC reduction in each case.

The highest IPC variation is caused by restoring the dependency of branches from previous instructions either through the flag register (imp. *flag-reg*) or through other general purpose registers (imp. *branch-regs*). These changes

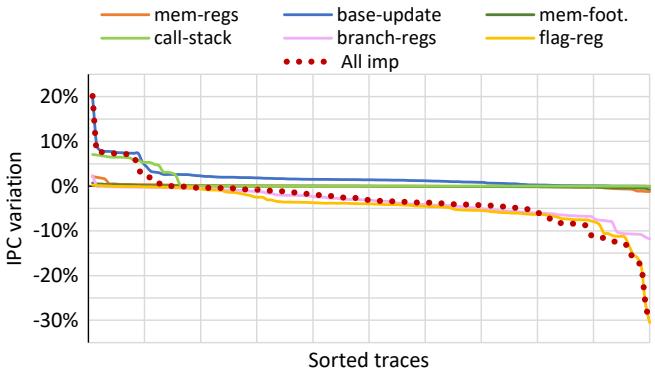

Figure 2: IPC variation when applying the different improvements to the *cvp2champsim* converter. Traces are sorted from highest IPC increase to highest IPC decrease for each improvement.

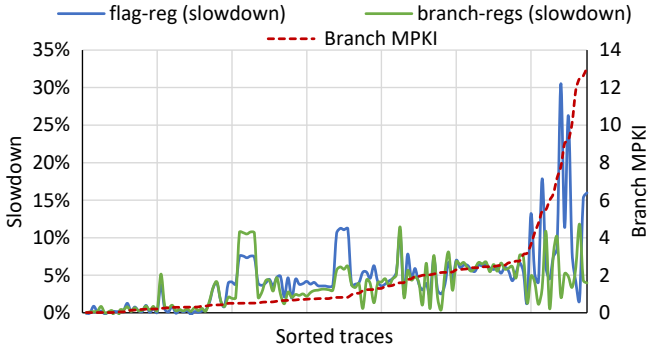

Figure 3: Slowdown due to improvements *branch-regs* and *flag-reg* compared to the original converter. Traces are sorted in increasing branch MPKI order (shown in the dashed line and right y-axis).

increase the impact of the branch misprediction penalties, as they are more easily exposed on the critical path. Consequently, they reduce the IPC: improvement *flag-reg* reduces IPC above 5% for 48 out of the 135 traces and up to 31% for trace *compute_int_46* while improvement *branch-regs* reduces IPC above 5% for 41 traces and up to 12% for trace *compute_int_23*. The trend of decreasing IPC is similar even though improvement *flag-reg* impacts performance more severely in the worst cases.

On the other hand, making the base register of memory operations available after the latency of an ALU instruction rather than after the memory access (imp. *base-update*) causes the highest IPC increases. This improvement accelerates the processor back-end, improving the IPC above 5% in 13 traces. The other improvement that improves performance noticeably is fixing the call stack (imp. *call-stack*). It affects only a subset of the traces, as not all of them had misclassified calls. This improvement reduces the branches MPKI and thus accelerates the processor front-end, increasing the IPC of 15 traces above 5%.

The remaining improvements have a minor impact on performance, at least in our ChampSim configuration with a decoupled front-end. Still, they make the converted traces more accurate and could impact performance in other processor configurations. Considering the proposed improvements altogether, the IPC of the converted traces differs by more than 5% from that of the original traces in 43 out of the 135 CVP-1 public traces. This showcases the importance of analyzing and understanding freely available traces and how well both the traces and the trace reader convey the characteristics of the original applications to a simulation infrastructure.

4.2. Discussion of Trace Improvements

Improvements *branch-regs* and *flag-reg*. With the original *cvp2champsim* converter, conditional and indirect branches missed almost all register dependencies with older instructions as branches only read from special purpose registers that non-branch instructions do not write. The improved converter restores these dependencies through improvements *branch-regs* and *flag-reg*, slowing down the processor front-end and reducing the IPC. The impact on performance, however, is not directly correlated to the number of dependencies restored. With a decoupled fetcher, the branch predictor and BTB, when accurate, hide the negative impact that delaying the execution of branches could have. Furthermore, other stalls at the front-end or back-end also affect whether the penalty of delaying branches is exposed or not.

Figure 3 presents the slowdown caused by improvements *branch-regs* and *flag-reg* compared to the traces converted with the original *cvp2champsim*. The traces are sorted from lowest to highest branch MPKI (plotted with the dashed line and right y-axis). As discussed previously, we cannot expect a perfect correlation. Yet, the Figure shows that as the branch MPKI of the traces grows, so does the slowdown caused by these improvements. In general, the slowdown caused by the two improvements is similar along the traces. However, there are some cases where improvement *flag-reg* has a significantly higher impact. These traces combine a higher percentage of ALU instructions without a destination register, to which improvement *flag-reg* adds the flag register as a destination, with a higher percentage of conditional branches reading from flags, which diminished the impact of improvement *branch-regs*.

Improvement *base-update*. Making the base register of memory operations available after the latency of an ALU operation rather than waiting for the memory operation to complete allows younger dependent instructions to execute earlier, accelerating the processor back-end and increasing performance. Figure 4 shows the speedup achieved by improvement *base-update*, with the traces sorted by increasing percentage of loads with base update compared to the dynamic number of executed instructions. We focus on loads and omit stores in this figure because they usually complete quickly (the actual write can be delayed in the store buffer but the destination registers become available to dependent instructions earlier) and do not delay dependent instructions as much as long-latency loads do. The figure shows that the speedup grows with the percentage of loads with base update, with a couple of exceptions where making

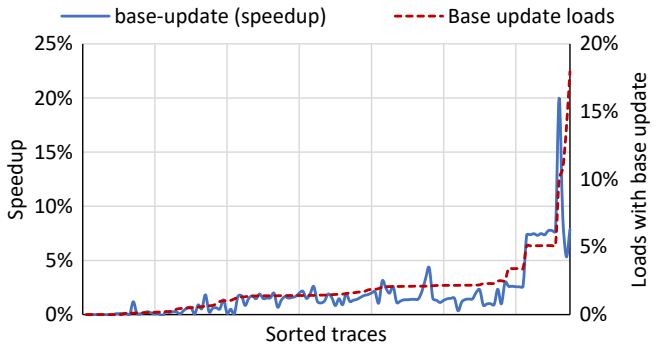

Figure 4: Speedup due to improvement *base-update*. Traces are sorted in increasing percentage of load instructions that perform base register update (dashed line and right y-axis) with respect to the overall number of instructions.

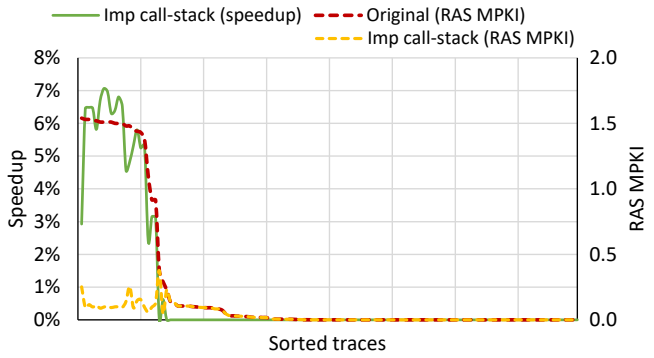

Figure 5: Speedup due to the *call-stack* improvement and RAS MPKI for the original and improved traces (dashed lines and right y-axis). Traces are sorted from highest to lowest RAS MPKI with the original traces.

the base register available earlier has a stronger impact even with a lower number of loads that perform base update.

Improvement *call-stack*. Figure 5 shows how the improved converter reduces the return's MPKI for the traces that suffered a high return MPKI with the original converter and the resulting IPC variation. As discussed in Section 3.2.1, branches reading and writing to register X30 should be classified as calls but were incorrectly identified as returns in the original converter. The figure shows that this issue does not affect all traces but only a subset of them which have a number of return mispredictions per kilo instruction one order of magnitude higher than the other traces. The improved converter brings back return target mispredictions to a reasonable number, which results in an IPC increase ranging approximately from 7% to 3%.

Improvement *mem-footprint*. An important limitation to accurately conveying the memory footprint of the original workloads to ChampSim is the absence of the memory access size in the ChampSim trace format. Access size would be required to accurately model memory disambiguation, including store-to-load forwarding. With this limitation, our *cvp2champsim* converter enhances the memory footprint of the converted traces by adding the memory address of the second cacheline when an access crosses cachelines. However, on average, only 0.3% of the instructions are memory instructions that access two cachelines. Given the low percentage and the facts that the second cacheline will be likely accessed anyway by a different instruction and that the access pattern is easily predictable for a simple next-line prefetcher, this improvement does not significantly affect performance in our setup.

Improvement *mem-regs*. This improvement affects 9.4% of the instructions, which are memory instructions with no destination register in the CVP-1 trace, to which our improved converter does not add any, and 5.2% of the instructions, which are loads with multiple destination registers that our improved converter now keeps. Based on the number of instructions affected, we could expect this improvement to have a significant impact on performance. However, this is not the case. Two reasons explain this behavior. First, many instructions to which the original *cvp2champsim* converter adds register X0 as destination appear in the trace *consecutively*, without any instruction that actually depends on them. Thus, only the last instruction introduces a spurious dependency, with negligible impact on performance. Nevertheless, notice that improving the traces in this aspect would be important if ChampSim modeled a finite physical register file. Second, keeping the original destination registers of the instructions does not have an always positive or negative impact on performance, as it depends on each specific case. For example, adding register X0 as a destination of an instruction that did not have it originally can make X0 available to dependent instructions earlier or later, depending on which was the previous instruction producing that register. The same thing happens when adding all destination registers to loads.

4.3. Characterization of IPC-1 traces

The mappings are disclosed in Table 2. The first column is the trace name in IPC-1 and the second column its corresponding (secret) trace in CVP-1. In the remaining columns, we perform a characterization of traces, once all fixes have been incorporated using the *develop* branch of ChampSim and the setup described in Section 4. For each trace, the table reports: IPC, branches MPKI, and MPKI across all the levels of the memory hierarchy. Regarding branches MPKI, we report the MPKI considering both the direction and target predictions (overall), considering only the direction prediction, and considering only the target prediction for taken branches.

The metric that sees the highest impact with the converter improvements is the target MPKI for branches, which reduces by 13%, on average, compared to the original traces. This reduction is caused by the *call-stack* improvement. As we have already discussed for the CVP-1 public traces, it mainly impacts some particular traces, whose branch target MPKI reduce by up to 78% (*server_001*). On average, IPC reduces by 2.4% and the IPC of 19 out of the 50 traces differs by more than 5% compared to the original traces. Because most of the performance difference comes from better conveying the register dependencies among instructions, the performance variations are not significantly

TABLE 2: CVP-1 TO IPC-1 TRACE MAPPING AND CHARACTERIZATION WITH THE IMPROVED CONVERTER.

			Branches MPKI			Memory hierarchy MPKI			
IPC-1 trace	CVP-1 trace	IPC	Overall	Direction	Target	L1I	L1D	L2	LLC
client_001	secret_int_294	2.37	2.56	2.11	1.54	10.0	19.5	8.3	3.2
client_002	secret_int_316	3.26	0.70	0.44	0.50	13.9	13.1	6.4	1.1
client_003	secret_int_729	2.11	2.55	1.82	1.70	14.3	29.1	8.3	3.9
client_004	secret_int_965	1.87	7.10	4.30	5.04	14.6	19.2	9.0	2.5
client_005	secret_int_349	1.82	3.16	2.15	2.13	16.9	22.0	9.4	5.5
client_006	secret_int_279	1.95	1.87	1.55	1.06	18.9	24.9	13.9	6.0
client_007	secret_int_591	2.49	2.16	1.41	1.45	25.7	25.4	5.8	2.8
client_008	secret_int_338	2.55	1.94	1.38	1.26	35.2	36.8	12.7	2.7
server_001	secret_srv160	2.25	0.43	0.31	0.30	16.8	22.1	13.9	6.3
server_002	secret_srv571	3.54	0.36	0.11	0.33	23.8	1.1	0.8	0.5
server_003	secret_srv757	1.48	4.84	3.25	3.38	29.7	28.9	30.0	5.7
server_004	secret_srv194	1.74	2.35	1.39	1.74	31.9	48.4	32.6	4.1
server_009	secret_srv551	2.17	0.93	0.44	0.73	36.8	39.0	36.4	2.9
server_010	secret_srv364	2.22	0.82	0.39	0.64	38.9	34.0	38.8	2.9
server_011	secret_srv617	1.92	2.19	1.33	1.60	39.5	25.6	39.1	3.8
server_012	secret_srv255	2.25	0.86	0.41	0.67	41.4	25.8	40.2	2.9
server_013	secret_srv442	2.16	0.89	0.44	0.69	43.0	26.1	43.1	3.3
server_014	secret_srv685	3.35	0.42	0.14	0.37	44.7	1.8	1.3	0.8
server_015	secret_srv238	3.78	0.22	0.04	0.20	46.2	0.4	0.3	0.2
server_016	secret_srv513	3.46	0.40	0.26	0.27	55.4	28.0	19.2	0.7
server_017	secret_srv155	0.61	0.71	0.69	0.59	64.1	129.3	52.4	25.7
server_018	secret_srv58	0.63	0.74	0.72	0.62	64.1	128.8	52.1	25.4
server_019	secret_srv564	0.61	0.67	0.64	0.57	64.6	129.8	52.8	26.0
server_020	secret_srv405	0.66	0.38	0.36	0.27	67.3	131.2	52.2	25.1
server_021	secret_srv174	0.70	0.20	0.18	0.13	68.5	133.4	52.6	24.4
server_022	secret_srv490	0.70	0.20	0.17	0.12	68.9	133.3	52.6	24.3
server_023	secret_srv152	3.31	0.53	0.34	0.37	73.1	37.3	27.4	1.0
server_024	secret_srv181	3.30	0.55	0.35	0.38	74.6	37.5	28.2	1.0
server_025	secret_srv301	3.44	0.45	0.24	0.33	76.3	36.4	29.9	0.7
server_026	secret_srv344	3.32	0.54	0.29	0.41	80.0	39.9	33.1	0.8
server_027	secret_srv428	3.43	0.47	0.25	0.35	81.1	38.8	31.0	0.7
server_028	secret_srv535	2.83	0.59	0.44	0.38	85.0	52.1	30.9	1.2
server_029	secret_srv91	2.82	0.58	0.45	0.37	85.8	52.5	29.9	1.3
server_030	secret_srv263	3.44	0.40	0.25	0.28	86.9	51.3	30.4	0.5
server_031	secret_srv656	2.78	0.62	0.48	0.40	89.2	47.1	27.1	1.3
server_032	secret_srv592	3.58	0.34	0.21	0.24	93.3	42.2	23.0	0.3
server_033	secret_srv7	3.45	0.14	0.12	0.09	98.6	21.3	7.5	0.9
server_034	secret_srv630	3.78	0.11	0.09	0.07	99.3	20.0	3.9	0.3
server_035	secret_srv374	2.42	0.11	0.08	0.07	98.2	21.9	8.8	4.3
server_036	secret_srv340	3.60	0.15	0.10	0.12	116.4	1.2	0.8	0.5
server_037	secret_srv680	3.28	0.17	0.13	0.14	116.8	14.9	6.9	0.8
server_038	secret_srv373	3.31	0.17	0.13	0.14	117.5	15.2	7.2	0.8
server_039	secret_srv154	3.83	0.14	0.06	0.13	121.8	2.0	0.1	0
spec_gcc_001	secret_int_118	1.88	7.79	6.62	4.53	10.6	19.4	6.2	1.9
spec_gcc_002	secret_int_345	0.20	0.52	0.48	0.28	16.4	178.6	131.4	78.2
spec_gcc_003	secret_int_123	0.16	0.35	0.31	0.19	21.2	143.1	136.3	96.2
spec_gobmk_001	secret_int_416	2.36	6.34	6.23	3.13	9.6	12.9	3.8	0.9
spec_gobmk_002	secret_int_121	2.39	6.89	6.82	3.50	12.9	3.4	1.1	0.7
spec_perlbench_001	secret_int_116	2.42	1.78	1.47	1.03	9.2	10.3	4.6	2.6
spec_x264_001	secret_int_919	3.59	1.20	1.19	0.62	8.5	4.4	0.9	0.6

reflected in the MPKI metrics. However, as a side effect of the *base-update* improvement, which increases the number of instructions in the traces, all MPKIs for the branches and memory hierarchy, are slightly reduced (1% – 4%).

4.4. Impact of the trace changes on IPC1

Having uncovered the mapping between CVP-1 and IPC-1 traces, we can perform a re-evaluation of the instruction prefetchers submitted to IPC-1 using the new traces.[4] The purpose of this evaluation is to check if the changes can affect the conclusions of a micro-architectural study, but *not to reflect the outcome of a hypothetical IPC-1 with improved traces*, as the submitted prefetchers were heavily tuned for the traces given at that time.

Table 3 shows the results of the IPC-1 championship (left part) and the results with the enhanced traces (right part). The prefetchers evaluated are D-JOLT [41], JIP [42], MANA [43], FNL+MMA [44], PIPS [45], EPI [46], BARÇA [47], and TAP [48]. This evaluation is performed using the ChampSim version employed in IPC-1, with the addition of the branch identification code modification described in Section 3.2.2.

4. This experiment is performed without the *mem-footprint* improvement because the version of ChampSim used at IPC-1 did not complete the execution of traces including instructions with multiple memory sources. Disabling this improvement should not introduce a significant performance deviation as it has a negligible impact on the current version of ChampSim.

TABLE 3: IPC-1 RANKING

Competition traces			Fixed traces		
Rank	Prefetcher	SpeedUp	Rank	Prefetcher	SpeedUp
1	EPI	1.2951	1	EPI	1.3818
2	D-JOLT	1.2884	2	D-JOLT	1.3696
3	FNL+MMA	1.2861	3	JIP	1.3588
4	BARÇA	1.2832	4	BARÇA	1.3570
5	PIPS	1.2799	5	FNL+MMA	1.3517
6	JIP	1.2768	6	PIPS	1.3444
7	MANA	1.2658	7	MANA	1.3092
8	TAP	1.2351	8	TAP	1.2915

Our first observation is that in general, the performance improvements are higher with the new traces. The main reason, as previously discussed in this evaluation, is the acceleration in the back-end due to the *base-update* improvement and the delay in the resolution of branches due to the *branch-regs* and *flag-reg* improvements. On one hand, accelerating the back-end puts more pressure on the front-end. On the other hand, resolving hard-to-predict branches later increases the misprediction penalty, which can be compensated with a good instruction prefetcher. The call-stack improvement does not influence the championship as the version of ChampSim used modeled an ideal target predictor.

Our second observation is that there are some differences in the hypothetical ranking with the improved traces. For example, JIP moves from the 6^{th} position to the 3^{rd}, which demonstrates that a careful translation of traces is fundamental for trustable research. Obviously, further prefetcher tuning could be done for the new traces, but this is out of the scope of this work. We also run the version of FNL+MMA submitted after the contest (same idea though additional tuning was performed), obtaining a speed-up with the fixed traces of 1.3812. That would move it to second place, which again demonstrates the importance of the tuning.

To conclude this study, we would like to emphasize Ishii et al. [49, 50], which highlighted the importance of modeling an industry-like decoupled front-end when evaluating instruction prefetching techniques. Results using a decoupled front-end would significantly reduce the performance improvements presented in this section due to the impact of including fetch-directed instruction prefetching [51] in the baseline. Our preliminary results are consistent with the previous work showing a much more modest speedup from dedicated instruction prefetchers from IPC-1 that we have been able to port to the new version of ChampSim. However, since these prefetchers have not been designed with a decoupled front-end in mind or other large changes to ChampSim since the IPC-1 contest, we decline to disclose those numbers here as they are of dubious value.

In that context, we recommend that a new instruction prefetching competition be held. The methodological problems pointed out by Ishii et al. as well as the inaccurate trace conversion we mitigate in this study call into question the magnitude of the contribution of the original IPC-1. Instruction prefetchers designed in using a now available more robust simulation environment could result in important and practical improvements in prefetching research state of the art.

4.5. Are the new traces more accurate than the original ones?

A fair but hard-to-address question is whether the traces generated with the improved converter reflect the original CVP-1 traces more accurately than the ones generated with the original *cvp2champsim* converter. Qualitatively, we have identified fundamental shortcomings and proposed solutions, giving us confidence that the overall quality of the converted traces has increased. In fact, improvements such as *call-stack* are a clear indicator that instruction categorization has improved. However, given the fact that the base material itself is flawed (CVP-1 traces have been stripped from some information), we cannot compare the converted traces against a ground truth (detailed ARMv8 traces), because that ground truth is not available to us.

5. Conclusion

At a broader level, this study highlights the need for carefully vetting the tools used to conduct research. Indeed, while we wholeheartedly advocate for sharing tools, some are built with very specific uses in mind and using them for a different purpose can lead to significant inaccuracies. It is reasonable for researchers to start with a tool that might have inaccuracies to get a good estimate of the potential of microarchitectural optimizations. Indeed, some of us who have worked in industry have observed that even the very detailed performance models used for exploration sometimes do not correlate with the baseline RTL they are intended to model. However, for the sake of pursuing realistic improvements in processor design, it behooves us to model the behavior of workloads and processors as faithfully as possible, continuously improving our methodology.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grants PID2021-123627OB-C51 and PID2022-136315OB-I00) and the European Union NextGenerationEU/PRTR (grants RYC2021-030862-I and TED2021-130233B-C33/C32), from the National Science Foundation (grants CNS-1938064 and CCF-1912617), as well as generous gifts from Intel. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

References

[1] "QEMU, A Generic and Open Source Machine Emulator and Virtualizer," https://www.qemu.org/.

[2] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, M. D. S. Boris Shingarov, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[3] 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, Oct. 2022.

[4] Standard Performance Evaluation Corporation, "SPEC CPU2006," 2006. [Online]. Available: http://www.spec.org/cpu2006

[5] ——, "SPEC CPU2017," 2017. [Online]. Available: http://www.spec.org/cpu2017

[6] "The 1st Championship Branch Prediction (CBP-1) @ MICRO," https://jilp.org/cbp/, Oct. 2004.

[7] "The 2nd Championship Branch Prediction (CBP-2) @ HPCA," Feb. 2006.

[8] "The 3rd Championship Branch Prediction (CBP3, JWAC-2) @ ISCA," https://jilp.org/jwac-2/program/JWAC-2-program.htm, Jun. 2011.

[9] "The 4th Championship Branch Prediction (CBP4, JWAC-4) @ ISCA," https://jilp.org/cbp2014/, Jun. 2014.

[10] "The 5th Championship Branch Prediction (CBP-5, JWAC-5) @ISCA," https://jilp.org/cbp2016/, Jun. 2016.

[11] "The 1st Cache Replacement Championship (CRC-1, JWAC-1) @ ISCA," https://jilp.org/jwac-1/, Jun. 2010.

[12] "The 2nd Cache Replacement Championship (CRC-2) @ ISCA," https://crc2.ece.tamu.edu/, Jun. 2017.

[13] "The 1st Data Prefetching Championship (DPC-1) @ HPCA," Feb. 2008. [Online]. Available: https://jilp.org/dpc/

[14] "The 2nd Data Prefetching Championship (DPC-2) @ ISCA," Jun. 2015. [Online]. Available: https://comparch-conf.gatech.edu/dpc2/

[15] "The 3rd Data Prefetching Championship (DPC-3) @ ISCA," Jun. 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/

[16] "The 1st Instruction Replacement Championship (IPC-1) @ ISCA," https://research.ece.ncsu.edu/ipc/welcome/, Jun. 2020.

[17] "The 1st Championship Value Prediction (CVP-1) @ ISCA," https://microarch.org/cvp1/cvp1online/rules.html, Jun. 2018.

[18] "The 1st Memory Scheduling Championship (MSC-1, JWAC-3) @ISCA," https://users.cs.utah.edu/ rajeev/jwac12/, Jun. 2012.

[19] "Ml prefetching competition @isca," https://sites.google.com/view/mlarchsys/isca-2021/ml-prefetching-competition, Jun. 2021.

[20] G. Vavouliotis, G. Chacon, L. Alvarez, P. V. Gratz, D. A. Jiménez, and M. Casas, "Page Size Aware Cache Prefetching," in *55thInt'l Symp. on Microarchitecture (MICRO)*. IEEE, 2022, pp. 956–974.

[21] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibánez, V. Viñals-Yúfera, and A. Ros, "Berti: an Accurate Local-Delta Data Prefetcher," in *55thInt'l Symp. on Microarchitecture (MICRO)*. IEEE, 2022, pp. 975–991.

[22] S. Jiang, Q. Yang, and Y. Ci, "Merging Similar Patterns for Hardware Prefetching," in *55thInt'l Symp. on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1012–1026.

[23] G. Vavouliotis, L. Alvarez, B. Grot, D. A. Jiménez, and M. Casas, "Morrigan: A composite instruction tlb prefetcher," in *54th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 1138–1153.

[24] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *48th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2021, pp. 99–111.

[25] S. Song, T. A. Khan, S. Mahdizadeh-Shahri, A. Sriraman, N. K. Soundararajan, S. Subramoney, D. A. Jiménez, H. Litz, and B. Kasikci, "Thermometer: Profile-guided btb replacement for data center applications," in *49th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2022, pp. 742–756.

[26] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *54th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 1121–1137.

[27] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *55th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2022, pp. 1–18.

[28] T. Asheim, B. Grot, and R. Kumar, "A storage-effective BTB organization for servers," in *29th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2023, pp. 1153–1167.

[29] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *45th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 519–532.

[30] B. Grayson, J. Rupley, G. D. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 40–51.

[31] A. Perais, "1st Championship Value Prediction Public Traces," https://doi.org/10.18709/perscido.2023.02.ds382, Jun. 2018.

[32] ——, "1st Championship Value Prediction Secret Traces," https://doi.org/10.18709/perscido.2023.02.ds384, Jun. 2018.

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *2005 Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[34] "The 2nd Championship Value Prediction (CVP-2)," https://microarch.org/cvp1/index.html, Feb. 2021.

[35] D. R. Kaeli and P. G. Emma, "Branch history table prediction of moving target branches due to subroutine returns," in *18th Int'l Symp. on Computer Architecture (ISCA)*, 1991, pp. 34–42.

[36] A. Perais, "CVP Trace Reader," https://gricad-gitlab.univ-grenoble-alpes.fr/tima/sls/projects/cvptracereader/-/tree/master, Oct. 2022.

[37] "ChampSim simulator," http://github.com/ChampSim/ChampSim, May 2020.

[38] A. Seznec, "A 64-Kbytes ITTAGE indirect branch predictor," in *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, Jun. 2011.

[39] ——, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Jun. 2016.

[40] I. E. Papazian, "New 3rd gen Intel® Xeon® Scalable processor (Codename: Ice Lake-SP)," in *32nd HotChips Symp.*, Aug. 2020, pp. 1–22.

[41] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-jolt: Distant jolt prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[42] V. Gupta, N. S. Kalani, and B. Panda, "Run-Jump-Run: Bouquet of Instruction Pointer Jumpers for High Performance Instruction Prefetching," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[43] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "MANA: Microarchitecting an Instruction Prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[44] A. Seznec, "The FNL+MMA Instruction Cache Prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[45] P. Michaud, "PIPS: Prefetching Instructions with Probabilistic Scouts," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[46] A. Ros and A. Jimborean, "The Entangling Instruction Prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[47] D. A. Jiménez, G. Chacon, N. Gober, and P. Gratz, "Barça: Branch Agnostic Region Searching Algorithm," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[48] N. Gober, G. Chacon, D. A. Jiménez, and P. Gratz, "Temporal Ancestry Prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[49] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Rebasing instruction prefetching: An industry perspective," *IEEE Computer Architecture Letters*, Oct. 2020.

[50] ——, "Re-establishing fetch-directed instruction prefetching," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2021, pp. 172–182.

[51] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *32nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 16–27.

[52] J. Feliu, A. Perais, D. A. Jiménez, and A. Ros, "Code artifact: Rebasing microarchitectural research with industry traces," https://zenodo.org/record/8265979, Aug. 2023.

[53] ——, "Data artifact: Rebasing microarchitectural research with industry traces," https://zenodo.org/record/8269409, Aug. 2023.

Appendix

1. Abstract

The key part of our artifact is the *cvp2champsim* converter, which implements all the improvements discussed in the paper. However, to allow the reproduction of our results, we also include the original CVP-1 public traces used in the paper. Finally, we include two versions of ChampSim. The first one is derived from the *main* branch at commit *2bba2bd* and includes the changes to branch type identification described in Section 3.2.2, a TAGE-SC-L prefetcher, and some additional statistics. The second one is the ChampSim version used for the IPC-1 contest with the evaluated prefetchers.

2. Artifact check-list (meta-information)

- **Compilation: gcc**
- **Data set:** CVP-1 public traces and IPC-1 traces.
- **Run-time environment:** Linux
- **Output:** All results presented in the paper.
- **How much disk space required (approx):** 500GB
- **Code license: Apache 2.0 License**
- **Data license: Creative Commons Attribution 4.0 International (CC BY 4.0)**
- **Archived: Code artifact: 10.5281/zenodo.8265979 Data artifact: 10.5281/zenodo.8269409**

3. Description

3.1. How to access. The artifact is divided into two parts: Code [52] and Data [53], which are permanently archived on Zenodo. In addition, we made a pull request on the ChampSim GitHub repository [37] to include the improved *cvp2champsim* converter.

3.2. Software dependencies. We run the experiments using Ubuntu 22.04 LTS. The *cvp2champsim* converter can be compiled with g++ version 7.5.0 (probably also with older versions). In addition, to compile the *main* version of ChampSim, we should install `make`, `curl`, and `pkg-config` (the latest two required by `vcpkg`). Compiling the IPC-1 version of ChampSim does not require any additional software. We provide scripts to launch the trace conversion and ChampSim executions sequentially and in parallel using `Slurm`, as well as scripts to gather the results presented in the paper. Some of these scripts use `python`.

3.3. Data sets. The data artifact includes the entire set of CVP-1 public traces and the subset of the CVP-1 secret traces used in the IPC1-1 championship. These are all the CVP-1 traces used in the paper.

4. Installation

First of all, we should extract the two artifact files and the Champsim compressed *tar*s included within the code artifact:

```
$ tar -xvf Artifact_code.tar
$ tar -xvf Artifact_data.tar
$ tar -xzvf ChampSim.tar.gz
$ tar -xzvf ChampSim-IPC1.tar.gz
```

From the root directory of the artifact, the *cvp2champsim* converter can be compiled with:

```
$ g++ cvp2champsim.cc -o cvp2champsim
```

To compile the *main* version of ChampSim, we should first install `vcpkg` and then configure the compilation with the `champsim_config_IISWC.json` file, which includes the processor setup used in the experiments:

```
$ cd ChampSim
$ ./vcpkg/bootstrap-vcpkg.sh
$ ./vcpkg/vcpkg install
$ ./config.sh champsim_config_IISWC.json
$ make
```

The IPC-1 version of ChampSim should be compiled setting each of the prefetchers evaluated in the competition. We provide a simple script that performs this process:

```
$ ./build_ipc1.sh
```

5. Experiment workflow

First of all, we should convert the traces with the improved *cvp2champsim* converter. The program template is:

```
$ ./cvp2champsim -t trace [-i improvement]
```

We can select the trace to convert and the improvements to apply, respectively, with the *–t* and *–i* options. The list of available improvements includes the individual improvements (i.e, *imp_mem-regs*, *imp_base-update*, *imp_mem-footprint*, *imp_call-stack*, *imp_branch-regs*, or *imp_flag-regs*), three sets of improvements (i.e., *All_imps*, *Memory_imps*, or *Branch_imps*), and no improvements (i.e., *No_imp*), which resorts to the conversion performed by the original converter. The converted trace will be written to the standard output. Converted traces are big but also compression-friendly.

For example, to convert *srv_0.gz* applying all improvements and, then, compress it with `xz`, we should run:

```
$ ./cvp2champsim -i All_imps -t ../CVP1_public_traces/srv_0.gz | xz -c > srv_0.champsimtrace.xz
```

The trace conversion is quick (1–3 minutes), but its compression with `xz` takes long (up to 1 hour).

To automatize the trace conversion, we provide the script `convert_traces.sh`:

```
$ ./scripts/convert_traces_seq.sh improvement suite [benchmark]
```

where improvement can refer to the same improvements listed before, and the suite can be either *CVP1public* or *IPC1*. The *benchmark* argument is optional: If a benchmark is indicated, only that benchmark will be converted; otherwise, all the benchmarks from the suite will be converted. This script converts traces sequentially and requires several hours to convert a suite.

To accelerate trace conversion, we also provide a `python` script that launches the trace conversion jobs in parallel using `Slurm`. The sequential script can be executed with:

```
$ ./scripts/convert_traces_paral.sh improvement suite
```

Finally, the script `convert_ALL_traces_paral.sh` takes no input parameters and launches, using `Slurm` all trace conversion jobs required to reproduce the results of the paper.

6. Evaluation and expected results

After the traces are converted, the ChampSim simulations should be launched. As done for the conversion, we provide scripts to launch them sequentially and in parallel using `Slurm`:

```
$ run-champsim-seq.sh ChampSim_binary improvement suite [benchmark]
$ run-champsim-paral.py ChampSim_binary improvement suite
```

For example, we can run all CVP-1 public traces with all trace conversion improvements using the *main* version of ChampSim with:

```
$ run-champsim-paral.py ChampSim/bin/champsim All_imps CVP1public
```

Unlike trace conversion, ChampSim executions are quick (a few minutes each) and do not have any disk space requirement. We also provide the script `run_ALL_champsim_paral.sh`, which takes no input parameter, to launch all the ChampSim executions needed to reproduce the results of the paper.

Finally, we provide a set of scripts to obtain all the results reproduced in the paper: `results_fig1.sh`, `results_fig2.sh`, `results_fig3.sh`, `results_fig4.sh`, `results_fig5.sh`, `results_tab1.sh`, and `results_tab2.sh`. The scripts assume that all required ChampSim executions are already completed. The scripts take no input parameter and give as output the data plotted in the corresponding figures.