Fast Interpreter-Based Instruction Set Simulation for Virtual Prototypes

Manfred Schlägl Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
manfred.schlaegl@jku.at daniel.grosse@jku.at

***Abstract*—The *Instruction Set Simulator*s (ISSs) used in *Virtual Prototype*s (VPs) are typically implemented as interpreters with the goal to be easy to understand, and fast to adapt and extend. However, the performance of instruction interpretation is very limited and the ever-increasing complexity of *Hardware* (HW) poses an increasing challenge to this approach.**

In this paper, we present optimization techniques for interpreter-based ISSs that significantly boost performance while preserving comprehensibility and adaptability. We consider the RISC-V ISS of an existing, SystemC-based open-source VP with extensive capabilities such as running Linux and interactive graphical applications. The optimization techniques feature a *Dynamic Basic Block Cache* (DBBCache) to accelerate ISS instruction processing and a *Load/Store Cache* (LSCache) to speed up ISS load and store operations to and from memory.

In our evaluation, we consider 12 Linux-based benchmark workloads and compare our optimizations to the original VP as well as to the very efficient official RISC-V reference simulator *Spike* maintained by *RISC-V International*. Overall, we achieve up to 406.97 *Million Instructions per Second* (MIPS) and a significant average performance increase, by a factor of 8.98 over the original VP and 1.65 over the *Spike* simulator. To showcase the retention of both comprehensibility and adaptability, we implement support for *RISC-V half-precision floating-point extension* (Zfh) in both the original and the optimized VP. A comparison of these implementations reveals no significant differences, ensuring that the stated qualities remain unaffected. The optimized VP including Zfh is available as open-source on GitHub.

I. Introduction

*Virtual Prototype*s (VPs) are high-level, executable models of *Hardware* (HW) platforms capable of running unmodified production *Software* (SW) [1], [2]. They are used to accelerate early design space exploration *before* physical HW is built. To ensure this acceleration is effective, VPs must be easy to create and understand. This is achieved by using high level languages like C++, domain specific standardized libraries like SystemC (IEEE 1666 [3]) [4]–[6], and abstraction of communication details with *Transaction Level Modeling* (TLM) [7]. For the same reason, although less efficient than dynamic binary translation methods, interpreter-based *Instruction Set Simulator*s (ISSs) are often preferred for simulating processors in VPs due to their ease of implementation, comprehensibility, and adaptability. Adding new instructions to an interpreter ISS typically involves straightforward modifications to the decoder and the instruction execution logic. However, their performance limitations often become apparent in later stages when VPs are used for interleaving HW and SW development or as reference models for verification. The techniques presented in this paper aim to significantly improve the performance of interpreter-based ISS implementations without sacrificing their comprehensibility or adaptability.

Contribution: Our contributions include two optimization techniques: (i) the *Dynamic Basic Block Cache* (DBBCache), which generates an alternative representation of the executed code, the *Dynamic Basic Block Graph* (DBBG), to efficiently cache data needed for instruction processing by the ISS, and (ii) the *Load/Store Cache* (LSCache), which allows direct translation of in-simulation virtual addresses to host system memory addresses, to speed up load and stores on data memory. In our evaluation, we compare these optimizations using 12 Linux-based benchmark workloads, achieving up to 406.97 *Million Instructions per Second* (MIPS) and a significant average performance increase, by a factor of 8.98 over an existing open-source VP and 1.65 over the efficient RISC-V reference simulator *Spike* from *RISC-V International* [8]. We also showcase that these optimizations retain comprehensibility and adaptability of the VP by implementing the *RISC-V half-precision floating-point extension* (Zfh) in both the original and optimized VP, with no significant differences observed. The optimized VP including Zfh is available on GitHub[1].

Related Work: There are a number of open-source RISC-V simulators, such as *RISC-V VP++* [9], *RISC-V VP* [10], *RISC-V-TLM* [11], *Spike* [8], *QEMU* [12], *RV8* [13] or *DBT-RISE* [14]. *RISC-V VP++*, its predecessor *RISC-V VP*, and *RISC-V-TLM* are all SystemC VPs with a non-optimized, interpreter-based ISSs. Due to its extensive functionality (see Section II), *RISC-V VP++* is chosen as the basis for this work. *Spike* comes with an already highly efficient caching interpreter-based ISS, and is therefore chosen as a comparison in our evaluation. *QEMU*, *RV8* and *DBT-RISE* use dynamic binary translation. Although dynamic binary translation offers higher performance, this work focuses on interpreter-based techniques for earlier motivated reasons. Commercial VPs, such as Synopsys Virtualizer, Siemens Vista and SIM-V from MachineWare are closed source, and thus exclude the qualities of *comprehensibility* and *adaptability* emphasized in this paper. To the best of our knowledge, the optimized VP resulting from this work has the highest performance interpreter-based ISS, among the SystemC-based, Linux-enabled VPs currently available as open-source.

II. Background and Outline

A key component of any HW platform is the processor. In recent years, the open, royalty-free RISC-V

[1]https://github.com/ics-jku/riscv-vp-plusplus

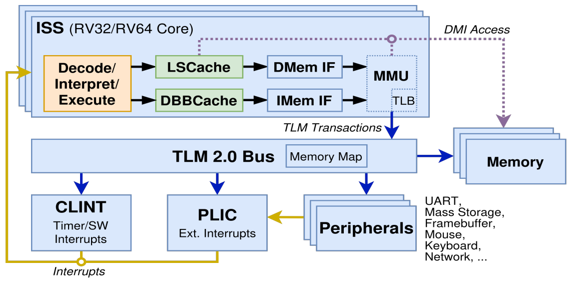


Fig. 1: *RISC-V VP++* Architecture

Instruction Set Architecture (ISA) [15], [16], known for its modularity and adaptability, has gained substantial traction in both academia and industry. A standout feature of RISC-V is its modularity, enabling extensive customization and specialization to balance performance with power efficiency. This flexibility is achieved through a range of standard extensions that can be added to the base ISA, enhancing the architecture's capabilities for specific tasks. One example of such an extension is Zfh, ratified in 2021 [17], which is used in our evaluation. Similar to the *F* and *D* extensions for single- and double-precision floating-point, Zfh adds support for half-precision floating-point to RISC-V.

This paper considers the open-source *RISC-V VP++* [9]. The VP provides extensive capabilities such as running Linux and interactive graphical applications [18], comes with support for the *RISC-V "V" Vector Extension* (RVV) [19], and is used for advanced verification approaches [20]–[22]. Its architecture is outlined in Fig. 1. The blue blocks show the components included in the original *RISC-V VP++*. The added and modified components presented in this paper are highlighted in green and orange, respectively. The VP includes interpreter-based ISSs for RISC-V in 32-bit (RV32) and 64-bit (RV64) configurations, and is capable of simulating multiple cores as indicated by the stacked ISS components in Fig. 1. The ISSs also include optional support for a *Memory Management Unit* (MMU) to realize *Virtual Memory Management* (VMM). A TLM-based bus links the ISSs, memory, and peripherals. The *CLINT* and *PLIC* provide timer and interrupt functionality. The VP offers several configurations, ranging from small platforms without VMM, e.g. for bare metal SW, to larger platforms with VMM, which can run OSes such as Linux.

The operation principle of the interpreter-based ISS can be seen in Fig. 2 [23]. At each iteration, an *Instruction Word* (*instrWord*) is fetched from the memory at the current *Program Counter* (PC) address. The *instrWord* is decoded to a ISS internal, unique *Operation Identifier* (*opId*), which is then used in a case distinction to select the appropriate implementation for the instruction. Finally, the chosen implementation extracts the necessary field values for execution (e.g., register addresses, immediate values, . . .) from the retrieved *instrWord*.

Static Basic Blocks (SBBs) are defined by the program structure and can be extracted with static code analysis. They start with a label to branch/jump to, and end with a branch/jump instruction. *Dynamic Basic Blocks* (DBBs), on the other hand, are determined dynamically by the actual execution flow and *Control Flow Changes* (CFCs), i.e. changes of the PC at taken branches, jumps, traps/interrupts and trap/interrupt returns [23]. The labels are not known in advance, so the start of a block is determined by the target PC of a CFC in the preceding block. The end of the block is determined by the next instruction in the execution flow that causes a CFC. In the **first optimization technique, DBBCache**, which we present in Section III, we identify and store DBBs during execution. For each instruction in such a block, we save all data that can be reused the next time the ISS processes the instruction. We also track all CFCs and store the relationships between blocks accordingly. This allows us to quickly switch to previously known blocks when the same CFC occurs in the future. As a result, we dynamically create an alternative representation of the executed code, which we call a DBBG.

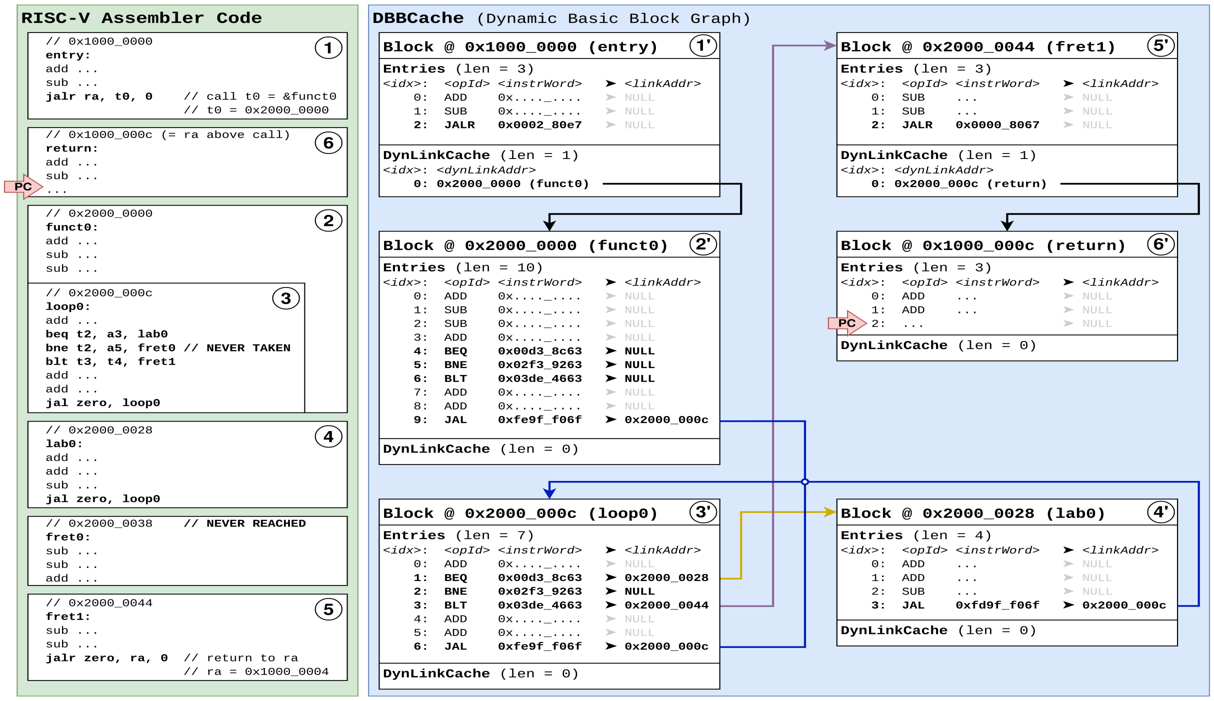
```
loop {
  (opId, instrWord) = fetch_decode(PC);
  switch (opId) {
    case ADD: regs[instr.rd] = regs[instr.rs1] + regs[instr.rs2]; break;
    case SUB: regs[instr.rd] = regs[instr.rs1] - regs[instr.rs2]; break;
    ...
  }
}
```

Fig. 2: ISS Instruction Interpretation

In SystemC VPs, *Direct Memory Interface* (DMI) is a technique to accelerate simulation by giving initiators a direct pointer to a memory area, thus bypassing more costly TLM transactions. Usually, in-simulation memory is realized by a contiguous host memory and is therefore DMI capable. The execution path of a RISC-V load/store instruction in *RISC-V VP++* is as follows: The ISS calls the *Data Memory Interface* (DMem IF) with the respective in-simulation address. If VMM is used, the in-simulation virtual address is translated to a in-simulation physical address using the MMU and the page table setup. The DMem IF checks, if the in-simulation physical address allows DMI. If so, the address is translated to a host system memory address, which is then directly dereferenced. Otherwise, a TLM transaction is issued. The goal of the **second optimization technique, LSCache**, which we present in Section IV, is to eliminate the above execution path for as many load/store instructions as possible. For all load/store instructions to DMI capable memory, we cache the host system memory addresses of the in-simulation page start addresses. As a result, subsequent accesses to cached in-simulation pages in the future can be realized directly by calculating and dereferencing host system memory addresses.

III. The Dynamic Basic Block Cache (DBBCache)

This section describes the DBBCache integrated in the ISS as seen in Fig. 1. Since each core can have its own VMM page table setup, each ISS instance has its own independent instance of a DBBCache. The DBBCache is organized in *Blocks*, which correspond to DBBs identified while execution. *Blocks* contain a variable number of *Entries* corresponding to the instructions covered by the DBB. *Entries* contain the *instrWord* and a decoded *opId* corresponding to the instruction implementation to execute in the ISS. New *Blocks* are created for the entry point, i.e. start PC of ISS execution, and any CFC in the execution flow. Each newly created *Block* is added to a *BlockHashMap*, to keep track of all *Blocks* and their start



```
// 0x1000_0000
entry:
add ...
sub ...
jalr ra, t0, 0      // call t0 = &funct0
                    // t0 = 0x2000_0000
```

```
// 0x1000_000c (= ra above call)
return:
add ...
sub ...
...
```

```
// 0x2000_0000
funct0:
add ...
sub ...
sub ...
```

```
// 0x2000_000c
loop0:
add ...
beq t2, a3, lab0
bne t2, a5, fret0 // NEVER TAKEN
blt t3, t4, fret1
add ...
add ...
jal zero, loop0
```

```
// 0x2000_0028
lab0:
add ...
add ...
sub ...
jal zero, loop0
```

```
// 0x2000_0038     // NEVER REACHED
fret0:
sub ...
sub ...
add ...
```

```
// 0x2000_0044
fret1:
sub ...
sub ...
jalr zero, ra, 0  // return to ra
                  // ra = 0x1000_0004
```

```
Block @ 0x1000_0000 (entry)
Entries (len = 3)
<idx>:  <opId> <instrWord>    ➤ <linkAddr>
    0:  ADD    0x...._....    ➤ NULL
    1:  SUB    0x...._....    ➤ NULL
    2:  JALR   0x0002_80e7    ➤ NULL
DynLinkCache (len = 1)
<idx>: <dynLinkAddr>
    0: 0x2000_0000 (funct0)
```

```
Block @ 0x2000_0000 (funct0)
Entries (len = 10)
<idx>:  <opId> <instrWord>    ➤ <linkAddr>
    0:  ADD    0x...._....    ➤ NULL
    1:  SUB    0x...._....    ➤ NULL
    2:  SUB    0x...._....    ➤ NULL
    3:  ADD    0x...._....    ➤ NULL
    4:  BEQ    0x00d3_8c63    ➤ NULL
    5:  BNE    0x02f3_9263    ➤ NULL
    6:  BLT    0x03de_4663    ➤ NULL
    7:  ADD    0x...._....    ➤ NULL
    8:  ADD    0x...._....    ➤ NULL
    9:  JAL    0xfe9f_f06f    ➤ 0x2000_000c
DynLinkCache (len = 0)
```

```
Block @ 0x2000_000c (loop0)
Entries (len = 7)
<idx>:  <opId> <instrWord>    ➤ <linkAddr>
    0:  ADD    0x...._....    ➤ NULL
    1:  BEQ    0x00d3_8c63    ➤ 0x2000_0028
    2:  BNE    0x02f3_9263    ➤ NULL
    3:  BLT    0x03de_4663    ➤ 0x2000_0044
    4:  ADD    0x...._....    ➤ NULL
    5:  ADD    0x...._....    ➤ NULL
    6:  JAL    0xfe9f_f06f    ➤ 0x2000_000c
DynLinkCache (len = 0)
```

```
Block @ 0x2000_0044 (fret1)
Entries (len = 3)
<idx>:  <opId> <instrWord>    ➤ <linkAddr>
    0:  SUB    ...            ➤ NULL
    1:  SUB    ...            ➤ NULL
    2:  JALR   0x0000_8067    ➤ NULL
DynLinkCache (len = 1)
<idx>: <dynLinkAddr>
    0: 0x2000_000c (return)
```

```
Block @ 0x1000_000c (return)
Entries (len = 3)
<idx>:  <opId> <instrWord>    ➤ <linkAddr>
    0:  ADD    ...            ➤ NULL
    1:  ADD    ...            ➤ NULL
    2:  ...                   ➤ NULL
DynLinkCache (len = 0)
```

```
Block @ 0x2000_0028 (lab0)
Entries (len = 4)
<idx>:  <opId> <instrWord>    ➤ <linkAddr>
    0:  ADD    ...            ➤ NULL
    1:  ADD    ...            ➤ NULL
    2:  SUB    ...            ➤ NULL
    3:  JAL    0xfd9f_f06f    ➤ 0x2000_000c
DynLinkCache (len = 0)
```

Fig. 3: DBBCache: Example for a *Dynamic Basic Block Graph* (DBBG) after Execution from `entry` to PC (red arrow)

PCs. However, as we will see later, the *BlockHashMap* is only used as a last resort when looking for a *Block* on a CFC.

An instruction is retrieved by the ISS by calling a function of the DBBCache with the current PC. This is called the fetch/decode step. The DBBCache checks if the next *Entry* in the currently active *Block* is available. If so, the instruction has already been processed and the cache has a hit. The cache advances to the subsequent *Entry* and returns its *opId* and *instrWord* to the ISS. Otherwise, the cache has a miss. The cache fetches the instruction from the instruction memory, decompresses potential compressed RISC-V instructions and decodes the instruction. The resulting *instrWord* and *opId* are stored in a new *Entry* in the currently active *Block*, and returned to the ISS. Finally, the ISS uses the *opId* to select the instruction implementation and the *instrWord* to extract the required field values.

An example of an identified DBBCache structure is shown in Fig. 3. Left, backed in green, we see the RISC-V assembler code of an executed code fragment including the execution entry, a call to a function with a loop and some branches, and a return from a function. The `add` and `sub` instructions are placeholders for any RISC-V instructions that can not result in a CFC. Right, backed in blue, we see the corresponding DBBG that was identified by the DBBCache during execution up to the PC indicated by the red arrow. The ISS starts at `entry` in ①. Correspondingly, the DBBCache identifies the DBB ①'. While the ISS continues to interpret the instructions from ① (`add`, `sub`, `jalr`), the DBBCache analogously creates *Entries* in ①'. The execution of the `jalr` instruction finally causes a CFC, resulting in the creation of DBB ②'. The CFCs and other important events of the ISS are reported to the DBBCache via provided interface functions. We will now first discuss the handling of CFCs in Section III-A. After that, we will look at other important events in more detail in Section III-B.

A. *Control Flow Changes (CFCs)*

In RISC-V, CFCs appear on taken branches, static and dynamic jumps, and trap/interrupt enters and returns. For didactic purposes, they will now be discussed in that order.

1) Taken Branches: If the condition of a RISC-V branch instruction is met, the control flow is changed by a constant offset relative to the PC, i.e. the branch is "taken". Otherwise, the execution continues with the next instruction, i.e. the branch is "not taken". For example, `beq t0, t1, 8` causes a CFC if the registers `t0` and `t1` are equal, in which case the PC is adjusted by the offset `8`.

The ISS reports each taken branch with the PC offset to the DBBCache. Each *Entry* contains a pointer to a *Block*, referred to as *linkAddr*, which is initialized with *NULL*. If the *linkAddr* of the current *Entry* is not *NULL*, the branch was taken before, which corresponds to a link hit. In this case, the cache switches to the pointed-to *Block* for the next fetch/decode step and returns to the ISS. Otherwise, the branch was never taken before, which corresponds to a link miss. In this case, the target PC is calculated using the offset and checked for existence in the *BlockHashMap*. If the PC was already identified as start of a *Block*, there is already a *Block* to switch to, otherwise a new empty *Block* is created. The *linkAddr* of the current *Entry* is set to the found or newly created *Block*, which allows to switch directly to this *Block* in the future. Finally, the cache switches to this *Block* for next fetch/decode step and returns to the ISS. Using this scheme allows us to have *Blocks* containing multiple branch instructions. This increases the overall average length of *Blocks* and with this the efficiency of the cache, because less block switches occur on average. Examples for branches are shown in Fig. 3. The code in ② and ③ defines a function with a loop containing three branches (`beq`, `bne`, `blt`). The corresponding DBBCache representations are shown in ②' and ③'. ②' contains the function's first loop iteration where none of the branches were taken, while ③' includes subsequent iterations, with two of the three branches (<`idx`> 1 and 3) taken at least once.

2) Static Jumps: A static jump in RISC-V changes the control flow by a constant offset relative to the PC. For example, `jal x0, -16` causes a CFC where the PC is adjusted by the offset `-16`. Static jumps are used to realize loops and (static) function calls. Static jumps are reported and handled in the same way as taken branches described above in Section III-A1 using the *linkAddr* in the current *Entry*. The only difference is, that static jumps are never *not taken*, which means, that they are always the last instruction in a *Block*. Examples for static jumps are shown in Fig. 3, at the end of ②, ③ and ④ (`jal`). The corresponding DBBCache representations can be seen in the last *Entries* of ②', ③' and ④'.

3) Dynamic Jumps: In contrast to static jumps, where the target PC is known at compile time, dynamic jumps in RISC-V allow to jump to locations calculated at run-time. For example, `jalr x0, t0, 4` causes a CFC where the PC is set to the value stored in register `t0` plus `8`. On RISC-V, dynamic jumps can be used to realize jumps over the full address range (i.e. wide jumps). Another common use of dynamic jumps is returning from functions. For example, a function is called with the static jump `jal ra, function`, where the PC of the following instruction, the return address, is saved in register `ra`. A return from the function is realized with the dynamic jump `jalr x0, ra, 0`, where the PC is set to the previously stored return address in `ra`. Other important use cases for dynamic jumps are, for example, function pointers in C or methods of polymorphic classes in C++.

The ISS reports each dynamic jump with the target PC to the DBBCache. Each *Block* contains a ring buffer, referred to as *DynLinkCache*. This *DynLinkCache* can hold up to 16 entries consisting of a identified target PC with a corresponding pointer to a *Block*, referred to as *dynLinkAddr*. If the target PC is found in the *DynLinkCache*, the dynamic jump to this PC has already been seen (link hit). The cache switches to the *Block* pointed-to by the found *dynLinkAddr* for the next fetch/decode step and returns to the ISS. Otherwise, the jump to that PC was never been seen (link miss). Again, the *BlockHashMap* is used to check whether a *Block* already exists or whether a new empty *Block* needs to be created. The PC and the pointer to the found or newly created *Block* is added to the *DynLinkCache*, which allows to switch directly to this *Block* in the future. Finally, the cache switches to this *Block* for next fetch/decode step and returns to the ISS. This generic approach is agnostic w.r.t the concrete use of the dynamic jump i.e. a complex algorithm to distinguish wide jumps, dynamic calls or returns is not necessary, which makes the handling very efficient. Examples of dynamic jumps can be found in Fig. 3, at the end of ① and ⑤ (`jalr`), once as a dynamic call and once as a function return. The corresponding DBBCache representations can be seen in the last *Entries* of ①' and ⑤'.

4) Trap/Interrupt Enters: Traps and interrupts are events that interrupt the normal execution flow of a program and cause switches to predefined handler code. Traps are synchronous to the execution flow and are triggered by the executed instruction, e.g. system calls, illegal instructions, access to invalid memory regions. Interrupts, are asynchronous to the instruction flow and are triggered externally, e.g. by peripherals, to indicate that they require attention.

RISC-V provides a wide variety of trap/interrupt handling schemes, ranging from handlers in different privilege levels to vectorized interrupt handling. For reasons of efficiency, we take a generic approach that is agnostic to all these schemes. The ISS reports all trap and interrupt enters with the corresponding target (handler) PC to the DBBCache. The DBBCache has a *TrapLinkCache* which holds up to 8 entries consisting of a identified target PC with a corresponding pointer to a *Block*, referred to as *trapLinkAddr*. If the target PC is found in the *TrapLinkCache*, the trap handler has already been entered (link hit). The cache switches to *Block* the pointed-to by the found *trapLinkAddr* for the next fetch/decode step and returns to the ISS. Otherwise, the trap handler has never been entered (link miss). Again, the *BlockHashMap* is used to check whether a *Block* already exists or whether a new empty *Block* needs to be created. The PC and the pointer to the found or newly created *Block* is added to the *TrapLinkCache*, which allows to switch directly to this *Block* in the future. Finally, the cache switches to this *Block* for next fetch/decode step and returns to the ISS.

5) Trap/Interrupt Returns: Returns from traps or interrupts cannot be handled in the same way as the function returns described in Section III-A3. Reasons for this are: (i) interrupts can happen at any point in execution, and (ii) both events may cause process context switches in an *Operating System* (OS). As a result, the return PC can point anywhere in program memory and does not necessarily indicate the start of a DBB. The likelihood of such *Blocks* reoccurring is very small, so caching them would be a waste of runtime and memory. When the ISS reports a trap or interrupt return to the DBBCache, the cache switches to a special, empty *DummyBlock*. Subsequent instructions are handled as cache misses, i.e. are fetched from memory and decoded. This continues until the next CFC in the instruction flow, whereupon the cache continues to work as described in the sections above.

B. Fences and Cache Coherency

Important events in the execution that need to be handled are changes that render the contents of the DBBCache potentially incoherent to the instruction memory. In case of the DBBCache, this can be (i) changes to the instruction memory contents (e.g. self-modifying code) or, (ii) changes to the VMM setup (e.g. page table changes). Since such changes usually also affect caches on real HW, most architectures provide special instructions which must be used by SW to indicate such changes. In RISC-V these are the fence instructions `fence.i` for (i) and `fence.vma` for (ii).

Each occurrence of one of these instructions is reported to the DBBCache by the ISS. The cache then increments a counter called *CoherenceCnt*. Each *Block* also has a corresponding *CoherenceCnt*, which is set to the value of the cache's *CoherenceCnt* when it is created or updated. In the fetch/decode step, the cache compares the *CoherenceCnt* of the current *Block* and the cache. If the counter values match, the *Block* is coherent and the cache continues. Otherwise, the *Block* may be incoherent and its *Entries* are compared against the memory contents. If there are no differences, the *Block* is still coherent. If there are differences, coherence is restored by updating the affected *Entries* from memory (fetch/decode) and by resetting the affected *Block* links (*linkAddr* and *DynLinkCache*). In both cases, the *CoherenceCnt* of the *Block* is set to that of the cache and execution continues.

C. DBBCache-Based Optimizations

Based on DBBCache and the high hit rates it can achieve, further optimizations are possible. We will now give a brief overview of some of the most important optimizations.

1) Computed Goto: In the ISS, we replace the *opId*-based case distinction with more efficient computed gotos, using macros to maintain clarity (e.g. Fig. 5 in Section V-B). These macros generate jump labels and create a compile-time lookup table for *opIds* to *jump label addresses*. After each instruction decode, the DBBCache uses this table to retrieve the *jump label address*, which is then stored in the *Entry* instead of the *opId* and returned to the ISS. The ISS then directly jumps to the appropriate instruction implementation.

2) Fast and Slow Path: The ISS and DBBCache contain several rarely entered code paths. For example, in the ISS this includes termination checks, interrupt, debug and trace handling. We move all these rarely used cases to a *slow path*, which is only taken if the corresponding functionality is requested, e.g. if an interrupt is set pending. In the DBBCache, we implement a *fast path* using a pointer to the currently active *Entry* which is simply incremented after each fetch/decode step. The end of *Block* is detected with a special end-of-block marker in the last *Entry* of a *Block*. The cache switches to the *fast path* on each CFC if the target *Block* is guaranteed to be coherent (Section III-B), and remains in *fast path* execution unless a cache miss or fence instruction occurs. The resulting *fast paths* are extremely lean and therefore highly efficient.

IV. The Load/Store Cache (LSCache)

Besides efficient instruction interpretation, another important factor for high ISS performance is data flow to and from memory. In this section, we describe the LSCache which is integrated into the ISS as seen in Fig. 1.

In RISC-V, load and store instructions exist for different types of data (e.g. different width integers, floats, ...). For example, `lw t1, t0, 8` loads a 32 bit word from a memory location pointed to by `t0` plus offset `8` into register `t1`.

As outlined in Section II, the goal of LSCache is to eliminate interaction with the DMem IF, including costly address translation by the MMU, DMI handling, etc., for as many load/store instructions as possible. The LSCache is organized as a direct-mapped cache with 256 entries and stores direct translations from in-simulation page start addresses to corresponding DMI host system memory addresses. From a given 64 bit in-simulation address, the cache uses the upper 44 bits $[63:20]$ as *TAG*, the following 8 bits $[19:12]$ as *Index* and the lowest 12 bits $[11:0]$ as *Offset* within the 4 KiB page. For each memory access, the cache checks, if the entry at *Index* is valid and its *TAG* matches the extracted *TAG*. If this is the case, the cache has a hit and directly dereferences the stored host system memory page address plus the *Offset* to perform the memory access. Otherwise, the cache has a miss, in which case the access is performed using the DMem IF as described above, but with one modification: For each DMI access, DMem IF stores the translated host system memory address. The cache checks whether the last access was using DMI. If not, the cache stays untouched. Otherwise, the cache retrieves the used host system memory page address, updates the cache entry at *Index* with the *TAG* and the retrieved host memory address and sets the entry valid. Similar to what is described for the DBBCache in Section III-B, a change in the VMM setup can cause the LSCache to become incoherent. Again, we use `fence.vma` to indicate such changes. Any occurrence is reported by the ISS to the LSCache, in which case all cache entries are invalidated.

The LSCache is agnostic to the use of VMM. It can directly translate from in-simulation physical or virtual addresses to host memory addresses for up to $256*4$ KiB $= 1$ MiB of DMI capable memory. It only takes a single cache miss anywhere in a 4 KiB page to eliminate DMem IF interactions for all subsequent accesses to the same page until the next VMM setup change.

V. Evaluation

In our evaluation, we first discuss in Section V-A the performance improvements of the introduced DBBCache and LSCache compared to the original *RISC-V VP++* and the *Spike* simulator. Then, in Section V-B, we demonstrate the preserved comprehensibility by comparing implementations of Zfh for both the original and the optimized VP.

A. Performance

All measurements were performed on a host system with an Intel® Core™ i7-10700 8-core processor running at 2.9 GHz, with 128 GiB RAM. The simulated *RV64* system is based on *Linux 6.9.0* [24], created by *buildroot-2023.08.2* [25] using the *GCC* compiler in version 13 [26].

Fig. 4 presents the results of our performance measurements and comparisons in three bar charts. The X-axis common to all charts shows the 12 selected workloads and the simulators examined. The top chart shows absolute results in MIPS obtained by taking the median of multiple measurement runs. The two charts below show the same results as acceleration factors relative to the original *RISC-V VP++* (middle) and the *Spike* simulator (bottom). As for the set of selected workloads, we use (i) *Dhrystone* [27] and *Whetstone* [28], targeting

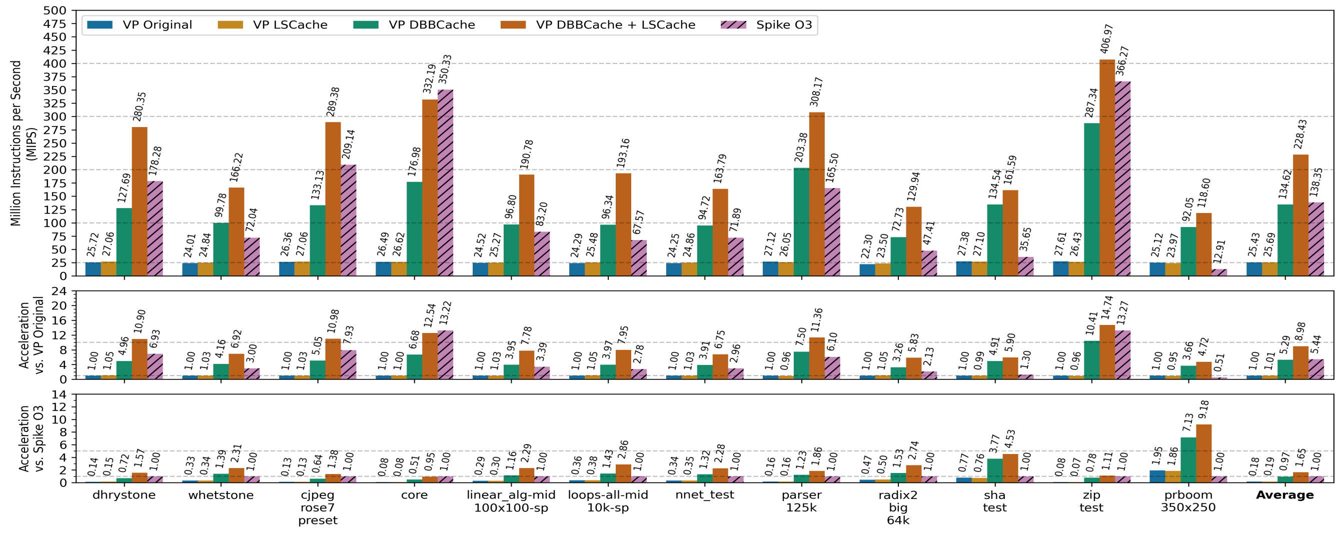


Fig. 4: Results of the Workload Performance Measurements and Comparison of the Simulators

integer and floating-point, respectively, (ii) all 9 workloads from *CoreMark®-PRO* [29], which target integer, floating-point and also the memory subsystem, and (iii) a two minute demo run of *PrBoom*, a Linux port of a classic game [30], rendering 350x250 images in a in-memory framebuffer, targeting integer, but also OS interaction. As for the simulators, we compare (Fig. 4, left to right) (i) the original *RISC-V VP++*, (ii) three optimized VPs with different combinations of DBBCache and LSCache enabled, and (iii) the *Spike* simulator, built with *GCC* optimization level 3.

We can see from Fig. 4 that using LSCache alone does not improve performance. The overhead of instruction interpretation is much higher than the overhead of accessing data. However, the combination of DBBCache and LSCache gives a significant performance boost compared to using only DBBCache. Another interesting observation is that *Spike* performs very well for most workloads, and even outperforms the optimized VP in *core*. However, in the case of *PrBoom*, the performance of *Spike* is conspicuously low. Since *PrBoom* interacts more frequently with the Linux kernel, a likely explanation is that *Spike* cannot handle context switches between kernel and userspace efficiently. A more detailed investigation is left for future work.

For the VP with DBBCache and LSCache enabled, we measure up to 406.97 MIPS for the *zip test* workload. On average, we get a significant performance increase by a factor of 8.98 over the original *RISC-V VP++* and 1.65 over *Spike*. In a run of all workloads with statistics enabled, the periodically calculated average hit rates never drop below 99.5% for the DBBCache and 98.7% for the LSCache. 98.5% of all executed instructions are handled in the ISS and DBBCache *fast paths*.

B. Preserved Comprehensibility and Adaptability

To demonstrate preserved comprehensibility and adaptability, we implement support for Zfh in both the original *RISC-V VP++* (*VP Original*) and the optimized VP (*VP DBBCache + LSCache*). The differences between these

VP Original

```
1  case OpId::FNMADD_H: {
2    fp_prepare_instr();
3    fp_setup_rm();
4    fp_regs.write(RD, f16_mulAdd(...
5    fp_finish_instr();
6  } break;
```

VP DBBCache + LSCache

```
1  OP_CASE(FNMADD_H) {
2    fp_prepare_instr();
3    fp_setup_rm();
4    fp_regs.write(RD, f16_mulAdd(...
5    fp_finish_instr();
6  } OP_END();
```

Fig. 5: ISS Implementation Differences for `fnmadd.h`

implementations are then used to assess whether the stated qualities remain unaffected by the introduced optimizations.

Both implementations change the same number of code lines, 530. 271 of these changes are related to the instruction decoding of Zfh. However, since decoding is not affected by the optimizations, there are no differences between the implementations, as expected. The remaining 259 changes are related to the interpretation and execution of Zfh. Here, we observe 68 differences, 2 for each of the 34 newly introduced instructions. These differences are related to the replacement of the case distinction by computed gotos as presented in Section III-C1. Fig. 5 shows an example of these differences for instruction `fnmadd.h` in Lines 1 and 6. We can see that the basic structure is maintained by the appropriately named macros, and therefore conclude that the optimizations presented in this paper have no significant negative impact on the comprehensibility or adaptability of the VP's ISS.

VI. Conclusions

In this paper, we presented in detail the two optimization techniques, DBBCache and LSCache, applicable to interpreter-based ISSs to significantly improve their performance while preserving comprehensibility and adaptability. These optimizations yielded a performance of up to 406.97 MIPS, with an average speedup factor of 8.98 over the unoptimized *RISC-V VP++* and 1.65 over the *Spike* simulator. The implementation of the RISC-V Zfh extension in both the original and optimized VPs showed no significant differences, confirming that the stated qualities remained intact. The optimized VP including Zfh is available on GitHub.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[2] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *Design, Automation and Test in Europe*, 2012, pp. 685–690.

[3] "IEEE 1666-2023 standard for standard SystemC language reference manual." [Online]. Available: https://doi.org/10.1109/IEEESTD.2023.10246125

[4] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[5] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[6] M. Hassan, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer, 2022.

[7] *OSCI TLM-2.0 Language Reference Manual*, OSCI, 2009. [Online]. Available: https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf

[8] "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim, 2024.

[9] M. Schlägl, C. Hazott, and D. Große, "RISC-V VP++: Next generation open-source virtual prototype," in *Workshop on Open-Source Design Automation*, 2024.

[10] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *Forum on Specification and Design Languages*, 2018, pp. 5–16.

[11] M. Montón, "A RISC-V SystemC-TLM simulator," 2020. [Online]. Available: https://arxiv.org/abs/2010.10119

[12] "QEMU a generic and open source machine emulator and virtualizer," https://www.qemu.org, 2024.

[13] "RV8," https://rv8.io, 2024.

[14] "DBT-RISE," https://github.com/Minres/DBT-RISE-Core, 2024.

[15] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and UC Berkeley, 2019.

[16] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and UC Berkeley, 2019.

[17] *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture, Version 20240411*, https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-698e64a-2024-09-09, RISC-V International, 2024.

[18] M. Schlägl and D. Große, "GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development," in *ACM Great Lakes Symposium on VLSI*, 2023, pp. 599–605.

[19] M. Schlägl, M. Stockinger, and D. Große, "A RISC-V "V" VP: Unlocking vector processing for evaluation at the system level," in *Design, Automation and Test in Europe*, 2024, pp. 1–6.

[20] C. Hazott and D. Große, "Relation coverage: A new paradigm for hardware/software testing," in *European Test Symposium*, 2024, pp. 1–4.

[21] M. Schlägl and D. Große, "Single instruction isolation for RISC-V vector test failures," in *International Conference on Computer-Aided Design*, 2024.

[22] C. Hazott, F. Stögmüller, and D. Große, "Using virtual prototypes and metamorphic testing to verify the hardware/software-stack of embedded graphics libraries," *Integr.*, vol. 101, p. 102320, 2025.

[23] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[24] "The Linux kernel archives," https://kernel.org, 2024.

[25] "Buildroot," https://www.buildroot.org, Accessed: 2023-02-19.

[26] "GCC the GNU compiler collection," https://gcc.gnu.org, 2024.

[27] "Dhrystone benchmark version 2.1," https://www.netlib.org/benchmark/dhry-c, 2024.

[28] "C converted Whetstone double precision benchmark version 1.2," https://www.netlib.org/benchmark/whetstone.c, 2024.

[29] "The EEMBC CoreMark-PRO processor benchmark," https://www.eembc.org/coremark-pro, 2024.

[30] "PrBoom," https://prboom.sourceforge.net, 2024.