# BasicBlocker: ISA Redesign to Make Spectre-Immune CPUs Faster

Jan Philipp Thoma
jan.thoma@rub.de
Horst-Görtz Institute
Ruhr-University Bochum
Bochum, Germany

Jakob Feldtkeller
jakob.feldtkeller@rub.de
Horst-Görtz Institute
Ruhr-University Bochum
Bochum, Germany

Markus Krausz
markus.krausz@rub.de
Horst-Görtz Institute
Ruhr-University Bochum
Bochum, Germany

Tim Güneysu
tim.gueneysu@rub.de
Horst-Görtz Institute
Ruhr-University Bochum
Bochum, Germany
DFKI GmbH, Cyber-Physical Systems
Bremen, Germany

Daniel J. Bernstein
djb@cr.yp.to
Horst-Görtz Institute
Ruhr-University Bochum
Bochum, Germany
University of Illinois at Chicago
Chicago, USA

## ABSTRACT

Recent research has revealed an ever-growing class of microarchitectural attacks that exploit speculative execution, a standard feature in modern processors. Proposed and deployed countermeasures involve a variety of compiler updates, firmware updates, and hardware updates. None of the deployed countermeasures have convincing security arguments, and many of them have already been broken.

The obvious way to simplify the analysis of speculative-execution attacks is to eliminate speculative execution. This is normally dismissed as being unacceptably expensive, but the underlying cost analyses consider only software written for current instruction-set architectures, so they do not rule out the possibility of a new instruction-set architecture providing acceptable performance without speculative execution. A new ISA requires compiler and hardware updates, but these are happening in any case.

This paper introduces *BasicBlocker*, a generic ISA modification that works for all common ISAs and that allows non-speculative CPUs to obtain most of the performance benefit that would have been provided by speculative execution. To demonstrate the feasibility of BasicBlocker, this paper defines a variant of the RISC-V ISA called BBRISC-V and provides a thorough evaluation on both a 5-stage in-order soft core and a superscalar out-of-order processor using an associated compiler and a variety of benchmarks.

## CCS CONCEPTS

• **Security and privacy** → **Side-channel analysis and countermeasures**; • **Computer systems organization** → *Architectures*.

## KEYWORDS

Spectre, Hardware, RISC-V

## 1 INTRODUCTION

The IBM Stretch computer in 1961 automatically speculated that a conditional branch would not be taken: it began executing instructions after the conditional branch, and rolled the instructions back if it turned out that the conditional branch was taken [9]. More sophisticated branch predictors appeared in several CPUs in the 1980s, and in Intel's first Pentium CPU in 1993 [24].

Software analyses in the 1980s reported that programs branched every 4–6 instructions [17]. Each branch needed 3 extra cycles on the Pentium, a significant cost on top of 4–6 instructions, especially given that the Pentium could often execute 2 instructions per cycle. However, speculative execution removed this cost whenever the branch was predicted correctly.

Subsequent Intel CPUs split instructions into more pipeline stages to support out-of-order execution and to allow higher clock speeds. The penalty for mispredictions grew past 10 cycles. Meanwhile the average number of instructions per cycle grew past 2, so the cost of each mispredicted branch was more than 20 instructions. Intel further improved its branch predictors to reduce the frequency of mispredictions. See generally [24].

Today the performance argument for branch prediction is standard textbook material. Accurate branch predictors are normally described as "critical" for performance, "essential", etc.; e.g., [11, 28, 31]. Deployed CPUs vary in pipeline lengths, but speculative execution is common even on tiny CPUs with just a few pipeline stages, and is universal on larger CPUs.

This pleasant story of performance improvements was then rudely interrupted by Spectre [35], which exploited speculative

behavior in various state-of-the-art CPUs to bypass critical security mechanisms such as memory protection, stealing confidential information via hardware-specific footprints left by speculatively executed instructions. This kicked off an avalanche of emergency software security patches, firmware updates, CPU modifications, papers proposing additional countermeasures targeting various software and hardware components in the execution flow with an impact on performance, and further attacks. The literature is reviewed below. Some countermeasures have been already broken, and it is difficult to analyze whether the unbroken countermeasures are secure.

## 1.1 Our Contributions

At this point the security auditor asks "Can't we just get rid of speculative execution?"—and is immediately told that this would be a performance disaster. *Every* control-flow instruction would cost $P$ cycles where $P$ is close to the full pipeline length, and would thus cost the equivalent of $P \times I$ instructions where $I$ is the number of instructions per cycle. This extra $P \times I$-instruction cost would be incurred every 4–6 instructions. The emergency security patches described above also sacrificed performance, but clearly were nowhere near *this* bad.

We observe, however, that this performance analysis makes an implicit assumption regarding the instruction set architecture. We introduce an ISA feature, BasicBlocker, that undermines this assumption. BasicBlocker is simple and can be efficiently implemented in hardware. We show how modifications to the compiler utilize the BasicBlocker design to minimize the performance penalty of removing not only branch prediction, but also speculative fetching (that is, instructions are fetched but never executed) from a processor. The resulting processor design is less complex than current speculative CPUs and therefore simplifies a CPU security audit.

To evaluate performance and demonstrate feasibility of Basic-Blocker, we start with an existing compiler and an existing CPU for an existing ISA; we modify all of these to support BasicBlocker; and we compare the performance of the modified CPU to the performance of the original CPU. We selected the RISC-V ISA [4] given its openness. To demonstrate the compatibility to different types of CPUs, we selected two implementation platforms, one in-order soft core (a CPU simulated by an FPGA) and a simulated superscalar out-of-order processor to allow evaluations without manufacturing a chip. Full details of our BBRISC-V ISA appear later in the paper.

The Spectre authors stated that they "*believe that long-term solutions will require fundamentally changing instruction set architectures*" [35]. Our performance results rely on a synergy between changes to the CPU and changes to the compiler, mediated by changes to the ISA. To improve deployability, we explain how a CPU supporting BasicBlocker can also run code compiled for the old ISA. Our protection against Spectre relies solely on a simple change to the CPU, namely disabling speculation, so it applies both to old code and to new code. Recompilation is necessary only for performance reasons to relieve occasional hot spots, not for security.

*Scope of This Work.* Beyond branch prediction, CPU designers have added many forms of speculation in the pursuit of every last bit of performance, and the only safe assumption is that every form of speculation threatens security. For example, there are now known attacks exploiting the prediction of return addresses [36] or speculative store-load forwarding [29].

BasicBlocker addresses specifically the performance loss of disabling *control-flow speculation*. This includes among others branch prediction and return-address speculation. Focusing on one form is essential to make the analysis tractable, and branch prediction in particular clearly qualifies as an important target. To protect against attacks exploiting other forms of speculation (e.g., "Spectre-STL"), we recommend that the CPU designer disable all forms of speculation, not just control-flow speculation. This is easy for any form of speculation with sufficiently small benefits, but otherwise raises ISA-design challenges and performance-analysis challenges.

## 1.2 The BasicBlocker Concept in a Nutshell

The $P$-cycle branch-misprediction cost mentioned above is the time from early in the pipeline, when instructions are fetched, to late in the pipeline, when a branch instruction computes the next program counter. If a branch passes through the fetch stage and is mispredicted, then the misprediction will not be known until $P$ cycles later, when the next program counter is computed. Every instruction fetched in the meantime needs to be rolled back.

The implicit assumption is that the ISA defines the branch instruction to take effect starting immediately with the next instruction. This assumption was already challenged by "branch delay slots" on the first RISC architecture in the 1980s; see generally [19]. A branch delay slot means that a branch takes effect only after the next instruction. The compiler compensates by moving the branch up by one instruction, if there is an independent previous instruction in the basic block, the contiguous sequence of instructions preceding the branch. A branch delay slot reduces the cost of a branch misprediction by 1 instruction, and the first RISC CPU pipeline was short enough that this removed any need for branch prediction.

A few subsequent CPUs used double branch delay slots, reducing the branch-misprediction cost by 2 instructions. Obviously one can define an architecture with $K = P \times I$ delay slots after each branch. However, code compiled for that architecture can only run on a processor with exactly $K$ delay slots. Since an optimal $K$ depends on the CPU, code would have to be compiled for every target CPU individually.

In a BasicBlocker ISA, there is a "basic block $N$" instruction guaranteeing that the next $N$ instructions[1] will all be executed consecutively. These instructions include, optionally, a branch instruction, which takes effect *after* the $N$ instructions, no matter where the branch is located within the $N$ instructions. The same ISA supports all values of $N$ simultaneously.

It is the CPU's responsibility to disable all speculative behavior, including speculative fetching. With BasicBlocker, a significant part of the performance lost from disabling control-flow speculation can be regained in many cases. The BasicBlocker ISA lets the compiler declare the basic-block size and move the branch up as far as possible within the block. The declaration of the basic-block size lets the CPU fetch all instructions in the basic block, without speculation.

---

[1]It is natural to consider a variant that counts $N$ fixed-length words (as an extreme, $N$ bytes) on an architecture with variable-length instructions.

If the branch instruction is not too close to the end of the block then the CPU can immediately continue with the next basic block, again without speculation. The overall performance benefit of this rescheduling for each basic block matches the benefit of whatever number of delay slots could be useful for that microarchitecture, without the disadvantage of having to be compiled differently for each number of delay slots. The new instruction further allows for tight integration of further optimizations such as hardware loop counters.

## 2 RELATED WORK

*ISA Modifications.* There is a long history of security features in ISAs including extensions to enforce control-flow integrity (CFI) [2, 18], memory protection (e.g. ARM-MTE [1]), or the flushing of microarchitectural states [59]. Other extensions simplify the secure implementation of complicated and security-critical aspects, e.g. by adding an instruction for AES computations [27]. All these ISA extensions introduce new instructions, that can be used by a programmer or compiler to harden a program against some specific attacks. Usage of the new features (and hence the protection) requires some modification of the binary (mostly through recompilation), but unmodified binaries run correctly as well. In all cases hardware changes are required to support the new instructions.

Some ISAs remove incentives for control-flow speculation, although not motivated by security. Berkley's Precision Timed (PRET) machines [38] target real-time computing applications which require a minimal worst-case runtime. Hence, control-flow speculation is substituted by a round-robin scheduling of instructions from different thread contexts. With BasicBlocker we focus on single-threaded applications to still perform well without control-flow speculation, but thread parallelism is likely to further improve performance. VLIW architectures [23] introduce instruction level parallelism by explicitly declaring instructions that can be executed in parallel at compile time. VLIW further uses compiler heuristics to make an educated guess about the direction of a branch. If the branch is resolved in a different direction, the compiler places compensating code at the branch target. This technique relocates the speculation problem to the compiler level. A major drawback of VLIW is the strict compiler dependency on the target platform: many microarchitecture decisions are embedded into the ISA, and code must be recompiled whenever those decisions change. BasicBlocker is carefully designed to not re-introduce speculation at compiler level and the code generated by the compiler does not depend on the microarchitecture of the target CPU. Furthermore, the experience with VLIW architectures has shown that it is hard to find enough instruction-level parallelism.

*Spectre Countermeasures.* Transient-execution attacks, including speculative-execution attacks, gained widespread attention after the disclosure of Spectre [35] and Meltdown [40]. Since then several attacks have shown many ways that transient execution can undermine memory protection and violate basic security assurances [12, 15, 35, 36, 40, 41, 47, 52–54, 57]. General surveys of attack vectors and countermeasures are provided by Kiriansy and Waldspurger [34], Szefer [48], and Cannella et al. [13, 14]. In the following we will focus on countermeasures against control-flow

speculation based attacks. Typically, such attacks arrange for mis-predicted instructions to access sensitive data. The instructions are eventually rolled back but still leave footprints in the microarchitectural state.

Some countermeasures prevent the attacker from controlling the branch prediction [51, 65]. Such countermeasures are specialized to prevent a specific type of Spectre attack in a specific setting. Other approaches close a specific covert channel, most prominently the timing channel introduced through caches [3, 10, 32, 33, 39, 45, 50, 55, 58–60, 64]. Again those countermeasures are targeted at a specific setting and other covert channels remain exploitable.

A more general approach of countermeasures targets the attackers ability to create a secret-dependent, transient CPU state in combination with a covert channel. This can be done by limiting the microarchitectural operations that can be performed on sensitive values [5, 46, 56, 61–63]. Such approaches require the knowledge which values are considered as secret as well as a model that defines which kind of behavior (instructions or group of instructions in a transient setting) is dangerous. The security and performance overhead is highly dependent on the selection of this security model and the definition is not trivial, as new channels are discovered constantly (e.g., [6]). Reported overheads reach from 10% [5] to 125% [56], but require the consideration of the specific measurement environment.

Like most of the mentioned countermeasures, BasicBlocker requires changes to the hardware mediated by the ISA. In contrast to other approaches, BasicBlocker does not aim to fix the problems induced by control-flow speculation, but rather tries to mitigate the performance penalty caused by removing control-flow speculation entirely. The reasoning behind this approach is that only the removal of speculative behavior is guaranteed to remove all speculation-based attack vectors, by removing speculation as root cause of the vulnerability. The comparability of the resulting performance overhead is limited, as we also consider the impact of speculative fetching, which is mostly ignored by state-of-the-art Spectre countermeasures.

This paper focuses on speculative-execution attacks. It should be possible to similarly address fault-based, transient-execution attacks by "preponing" fault detection, removing most of the performance benefit of transient execution after faults, but further investigation of this idea is left to future work.

## 3 SPECULATION IN PROCESSORS

In a pipelined processor, each instruction passes through multiple pipeline stages before it eventually retires. A textbook series of stages is *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execution* (EX), *Memory Access* (MEM) and *Write Back* (WB) [49]. More complex CPUs can have many more stages.

If each stage takes one cycle then a branch instruction will be fetched on cycle $n$ in IF, decoded on cycle $n + 1$ in ID, and executed on cycle $n + 2$ in EX, so at the end of cycle $n + 2$ the CPU knows whether the branch is taken or not. Without branch prediction, IF stalls on cycles $n + 1$ and $n + 2$, because it does not know yet which instructions to fetch after the branch. With branch prediction, IF speculatively fetches instructions on cycles $n + 1$ and $n + 2$, and ID speculatively decodes the first of those instructions on cycle

$n + 2$. If the prediction turns out to be wrong then the speculatively executed instructions are *rolled back*: all of their intermediate results are removed from the pipeline.

The functional effects of instructions are visible only when the instructions retire, but side channels sometimes reveal microarchitectural effects of instructions that have been rolled back. As Spectre illustrates, this complicates the security analysis: one can no longer trust a branch to stop the wrong instructions from being visibly partially executed.

The standard separation of fetch from decode also means that *every instruction is being speculatively fetched*. An instruction fetched in cycle $n$ could be a branch (or other control-flow instruction), but the CPU knows this only after ID decodes the instruction in cycle $n + 1$, so IF is speculatively fetching an instruction in cycle $n + 1$. We emphasize that this behavior is present even on CPUs without branch prediction: the CPU cannot know whether the instruction changes the control flow before decoding it.

Disabling all control-flow speculation execution thus means that every branch must stall fetching until it is executed, and, perhaps even more importantly, that *every instruction* must stall fetching until it is decoded. BasicBlocker addresses both of these performance problems, as shown below.

## 4 CONCEPT

In this section, we outline the rationale behind our approach as well as the modifications to the ISA that allow the elimination of control-flow speculation within the microarchitecture. Though we use the RISC-V instruction set in the following examples, our solution is generally applicable to any ISA or processor as motivated in Section 4.4 and 4.5.

### 4.1 Design Rationale

It is conceptually simple to thwart security issues arising from control-flow speculation by entirely removing it, but is generally believed to incur a severe loss in performance. BasicBlocker addresses this by providing metadata through an ISA modification to assist non-speculative hardware with efficient execution of software programs.

The CPU has a limited view of programs, accessing only a small number of instructions at a time. With current ISAs, control-flow instructions appear without advance notice, and their result is available only after multiple pipeline stages, even though this result is needed immediately to infer the next instruction.

BasicBlocker takes the concept of basic blocks to the hardware level using novel instructions (in contrast to the textbook definition, we require a basic block to be terminated by all control-flow instructions, i.e. also calls). At compile time a holistic view of the program is available in form of a control-flow graph, including code structures such as basic blocks and control-flow changes. BasicBlocker uses the information available at compile time, specifically the length of individual basic blocks, and makes it available to the CPU during execution. This allows a non-speculative CPU to avoid most pipeline stalls, through the advance notice of control flow changes.

### 4.2 Basic Block Instruction

We introduce a new instruction, called basic block instruction (bb), which lays the foundation for BasicBlocker. Currently, most CPUs use control-flow speculation to gain performance. Enabling fast but non-speculative fetching requires additional information for the CPU, since normally we know that we can fetch the next instruction only after the prior instruction was decoded and it is ensured that the control flow does not deviate. Hence, normally the fetch unit would have to be stalled until the previous instruction was decoded. To avoid that delay, we define a new bb instruction that encodes the size of the basic block. Within this basic block, the CPU is allowed to fetch instructions, knowing that upcoming instructions can be found in sequential order in memory and will definitely be executed. That is, since per definition no control-flow changes can occur within the basic block. The instruction further provides information whether the basic block is *sequential*, stating that the control flow continues with the next basic block in the sequence in memory, i.e. the block does not contain a control-flow instruction. Figure 1 shows the transformation of traditional code (left) to code with bb instructions (right). The fetch unit of the CPU is responsible for counting the remaining instructions in a given block and only fetch until the end of the basic block. From there, the program continues executing the next basic block which itself starts with a bb instruction.

We also modified the behavior of existing control-flow instructions, such as bne, j and jlre. The goal is to give advance notice of upcoming control-flow changes to the CPU. Since the processor knows the number of remaining instructions per basic block, we can schedule control-flow instructions within basic blocks as early as data dependencies allow, and still perform the change of the control flow at the end of the basic block. This key feature allows the CPU to correctly determine the control flow before the end of the basic block, and renders branch prediction in many cases obsolete.

As a result, the only time that the CPU needs to stall fetching is at the transition of two basic blocks, because the following bb instruction needs to be executed before knowing the size and, hence, being able to continue fetching. To avoid this delay, it is sufficient to add the capability of representing one additional set of basic block information internally and request this information as early as possible. This means that the CPU interposes the bb instruction of the next basic block as soon as the next basic block is known, regardless whether there are instructions left in the current basic block or not.

In Figure 2, this principle is illustrated for the code of Figure 1 (right side). The bb instruction of the second basic block is fetched as soon as the branch target of bne is known. Afterwards, the execution of the first basic block continues. Execution of the second basic block can start as soon as the first basic block is consumed and the size of the second basic block is known (after EX of bb). If the current basic block does not contain a control-flow instruction, which is indicated by the *sequential* flag of the bb instruction, the CPU can fetch the next bb instruction directly. Otherwise, the next bb instruction will be fetched after the control-flow instruction passes the execution stage.

While the early fetching of the bb instruction changes the execution order, it does not affect security or correctness since the instruction is only fetched after the execution path is known for certain.

```
; Start of first basic block          bb     6, 0 ; first bb, size = 6, not seq
add     a5,a0,a4                       add    a5,a0,a4
add     t4,a3,a4                       add    t4,a3,a4
addi    a4,a4,8                        addi   a4,a4,8
mul     a1,t3,t2                       bne    a4,a6,80... ; compute branch result
lh      t2,0(a5)                       mul    a1,t3,t2
bne     a4,a6,80... ; compute branch and change PC    lh     t2,0(a5) ; change PC after this instr.
; Start of 2nd basic block             bb     2, 1 ; 2nd bb, size = 2, seq
lh      a7,0(a1)                       lh     a7,0(a1)
li      a4,0                           li     a4,0
; Start of 3rd basic block             bb     16, 0 ; 3rd bb, size = 16, not seq
sh      a1,0(a0)                       sh     a1,0(a0)
...                                    ...
```

**Figure 1: Example code for the new bb instruction. Left: Traditional RISC-V code does not contain information about the size of upcoming basic blocks. The bne instruction terminates the first block and conditionally branches. Right: The bb instruction gives information about upcoming code parts. The first basic block is terminated by the size given in the line 1 and performs a conditional branch based on the outcome of the bne instruction, whose result is already determined earlier.**
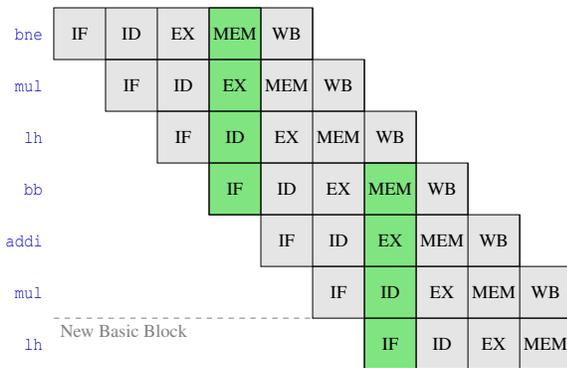


**Figure 2: Pipeline diagram for optimal code. The bb instruction of the next basic block is fetched as soon as the branch was executed. The branch only takes effect at the end of the current basic block. When the branch instruction is sufficiently early rescheduled, the next basic block can be fetched without stalls.**



**Figure 3: The worst case scenario has a branch instruction at the end of a basic block.**

Even with these changes it is sometimes necessary to stall the CPU at the transition of two basic blocks until the size of the new basic block is known. Therefore, this concept works best with software that contains many large basic blocks with opportunities to *reschedule* control-flow instructions at compile time. Software with a large number of small basic blocks is therefore less efficient, leading to pipeline stalls.

The worst case is a control-flow instruction that could not be rescheduled, since then the CPU needs to be stalled both for the information from the control-flow instruction as well as from the bb instructions. This case is depicted in Figure 3. We address the performance impact of small basic blocks in Section 4.3.

Overall, the rescheduling concept can be imagined as a variably-sized branch delay slot. There are two core advantages of our concept over traditional branch delay slots:

- The CPU does not need special constructs for the branch delay instructions. At the end of a basic block, the CPU can simply fetch the instruction at the target address, regardless
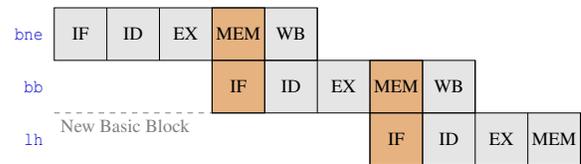
of the type of instructions that were executed prior. If the basic block was sequential, the target register defaults to $PC + 4$. If any control-flow operations were executed, the target register points to the target address.

- By having a variably-sized branch delay mechanism, the code is compatible to all hardware architectures that support the bb instruction. Since the control-flow instructions were rescheduled as early as possible, the code is optimal for those hardware architectures. For fixed size branch delay slots, CPUs with smaller pipelines may introduce unnecessary nop instructions.

See also Section 4.3 for further optimizations that integrate tightly with the bb instruction.

*4.2.1 ISA-Extension Specification.* We now define the changes required by BasicBlocker more precisely. A processor supporting the bb instruction is required to have an instruction counter $IC$, a target register $T$, a branch flag $B$, and an exception flag $E$, all initialized to 0 on processor reset and used only as defined below. The functional behavior of the bb instruction is given in Definition 4.1, the changes to the control flow in Definition 4.2 and the behavior that raises an exception in Definition 4.3.

*Definition 4.1 (BB Instruction).* The bb instruction takes a size parameter $n > 0$ and a sequential flag *seq*, and is executed as follows. If $IC = 0$: $IC \leftarrow n$; if *seq* $= 0$ then $B \leftarrow 1$; if *seq* $= 1$ then $B \leftarrow 0$ and $T$ is set to the address of the $n + 1$-th instruction following the bb instruction. Otherwise, if $IC \neq 0$, then $IC \leftarrow 0$ and $E \leftarrow 1$ to catch illegal bb instructions.

Thus, on a functional level, Definition 4.1 only sets $IC$, $T$, $B$, and $E$ but has no further effect on the execution of a program. The subsequent definitions have further effects.

*Definition 4.2 (BB-Delayed Branches).* The execution of non-bb instructions is modified as follows:

- Before every non-bb instruction: if $IC > 0$ then $IC \leftarrow IC - 1$.
- During every control-flow instruction: any write to $PC$ is instead written to $T$ if $B > 0$, and is ignored if $B = 0$.
- After every control-flow instruction: if $B = 0$ then $E \leftarrow 1$; otherwise $B \leftarrow B - 1$.
- Subsequently, after every non-bb instruction: if $IC = 0$ then $PC \leftarrow T$; and if $IC = 0$ and $B > 0$ then $E \leftarrow 1$.

BasicBlocker raises an exception ($E = 1$) whenever the bb instruction is used in an illegal way.

*Definition 4.3 (BB Exceptions).* After every instruction, an exception is raised if $IC = 0$ and $E \neq 0$.

In other words, after the $n$ instructions covered by a bb instruction, an exception is raised if any of the following occurred:

- $seq = 0$ and there was not exactly one control-flow instruction in the $n$ instructions;
- $seq = 1$ and there was a control-flow instruction in the $n$ instructions.
- A bb instruction appears within the $n$ instructions indicated by the previous bb instruction.

All three definitions are required, in order to add BasicBlocker to an arbitrary ISA. The following extra requirement, a requirement to use bb instructions, slightly simplifies the implementation of BasicBlocker, although later we consider dropping this requirement for compatibility.

*Definition 4.4 (Enforced BB).* In a BasicBlocker CPU with enforced BB: Before every non-bb instruction (and before $IC$ is decremented), an exception is raised if $IC = 0$.

To achieve an increased performance, an implementation of BasicBlocker can pre-execute bb instructions (cf. Figure 2) as defined in Definition 4.5. This pre-execution affects the microarchitecture and timing but not the ISA semantics.

*Definition 4.5 (BB Prefetching).* A BasicBlocker CPU with prefetching pre-executes a bb instruction $bb_{i+1}$ during the execution of a block, indicated by the bb instruction $bb_i$, as soon as:

- if $bb_i$ is sequential: $bb_i$ is resolved.
- if $bb_i$ is not sequential: the first control flow instruction of the block is resolved.

This requires an additional register $P$ which holds the values $n$ and $seq$ until execution reaches the instruction following the prefetched bb instruction. More precisely, when $IC = 0$ and $E = 0$:

- $IC \leftarrow n$ taken from $P$.
- if $seq = 0$ in P than $B \leftarrow 1$ else $B \leftarrow 0$.

If the prefetch address is invalid, or if the prefetch address is valid but the prefetched instruction is not a bb instruction, then pre-execution is skipped and does not raise an exception.

## 4.3 Further Optimizations

The above presented concept can be further optimized by providing the information contained in the bb instruction as soon as possible using pipeline forwarding. By construction, none of the information contained in the bb instructions affects any other element of the CPU than the fetch unit. Hence, it is possible to wire these bits back to the fetch unit directly after the decode stage without further changes to the design. Another clock cycle can be saved by using a bit mask to fast-decode the output of the instruction memory directly, with only marginal overhead.

A significant boost for performance can be achieved by introducing an additional interface to the instruction memory (or cache) that is used to access bb instructions. This would allow the fetch unit to request and process bb instructions in parallel with the normal instructions and, therefore, eliminate the entire performance overhead that is introduced though the addition of these instructions. Since a basic block contains always at least one instruction additional to the bb instruction, this instruction can be fetched before knowing the size of the basic block, without violating the above stated principles.

Further optimizations are possible with additional changes to the ISA. For example, the 1-bit sequential flag can be replaced by a multi-bit counter of the number of control-flow instructions in the upcoming block, so (e.g.) `if(a&&b&&c)` can be expressed as three branches out of a single block. This also changes the branch flag $B$ to a multi-bit branch counter.

The idea to announce upcoming control-flow changes early on is also the foundation of hardware loop counters, as already discussed in the literature [20, 44]. Here, the software announces a loop to the hardware, which then takes responsibility for the correct execution. We can seamlessly support hardware loop counters in our design concept. One new instruction (`lcnt`) is necessary to store the number of loop iterations into a dedicated register. The start and end address of a loop can be encoded into the bb instruction, by indicating with two separate flags whether the corresponding basic block is the start (s-flag) or end (e-flag) block of the loop. This allows the hardware to know the next basic block, as soon as the bb instruction of the end block gets executed. The fast execution of nested loops can be supported by adding multiple start and end flags to the bb instruction as well as adding multiple registers for the number of loop iterations. A more detailed description of the loop counter integration to our concept can be found in Appendix A.

## 4.4 Compatibility

For simplicity and comprehension all examples above consider an in-order, single issue processor with a generic five stage RISC pipeline. Control-flow speculation is widely used in such processors: e.g., the ARM Cortex-A53, which has shown to be vulnerable against speculative-execution attacks [42]. There is also tremendous interest in larger, super-scalar, out-of-order processors, where control-flow speculation is universal.

Adding support for out-of-order processors is trivial as per design, every instruction that is fetched by the processor will be retired - that is, if none of the instructions raise an exception. Once the CPU fetches the instruction, reordering is permitted as far as functional

correctness is ensured. Utilizing the two counter sets, reordering can be done beyond basic block borders if the bb instruction of the following basic block has been executed.

Similarly, support for superscalarity is easy to achieve. Once the bb instruction is executed, the CPU may fetch and execute all instructions within the current basic block in an arbitrary amount of cycles. If the successor basic block is known the CPU may fetch instructions from both basic blocks in one cycle. Secondary pipelines may also be useful to pre-execute bb instructions for the following basic block in parallel as described earlier.

Generally, the pipeline length can be chosen flexibly. However, as the CPU needs to wait for results of branch and bb instructions, it is desirable to make the results of these instructions available as early as possible.

A major feature of modern systems is the support of interrupts and context switches. We note that our concept does not impede such features; it merely increases the necessary CPU state that needs to be saved in such an event. More specifically, it is necessary to save the already gathered information about the current and upcoming basic blocks as well as the state of the loop counter, in addition to all information usually saved during a context switch. It is important that this data is secured against manipulation but that is true for all data stored during a context switch (e.g. register values, FPU state, ...).

Our proposal includes one new instruction and a modification to existing control-flow instructions. For easier deployability, it is desirable for a BasicBlocker CPU to be backwards-compatible. One could define new BasicBlocker control-flow instructions separate from the previous control-flow instructions. However, it suffices to interpret a control-flow instruction as having the new semantics if it is within the range of a bb instruction, and otherwise as having the old semantics, dropping Definition 4.4. Legacy code compiled for the non-BasicBlocker ISA will then run correctly but with low performance, and code recompiled to use bb will run correctly with high performance.

It would also be possible to integrate our solution into a secure enclave by providing a modified fetch unit for the enclave. Security critical applications could run in the protected enclave while legacy software can be executed on the main processor without performance losses.

We do not expect major problems for implementing BasicBlocker in JIT compilers. In some cases, JIT compilers already need to know the basic-blocks of the program and in general one can artificially insert sequential basic-blocks according to the window size of the compiler.

## 4.5 BasicBlocker for Generic ISAs

In the following, we outline the changes necessary to implement the BasicBlocker concept in arbitrary ISAs. We observe that in common ISAs, branches are realized with three basic operations which are performed by a varying number of instructions.

(1) First, the operands on which the branch decision will be made are compared. The result of the comparison may be saved in a special purpose flag (e.g. Intel x86, ARM), a register value, or used immediately (e.g. RISC-V, some Intel x86).

(2) Based on the outcome of the comparison, the target address is computed.

(3) The instruction pointer is changed to the target address computed in the previous stage.

For most ISAs, steps 2) and 3) are combined to one instruction. RISC-V is unusual in having *only* branch instructions that combine all three operations.

A BasicBlocker ISA is required to separate operation 1) and 2) from 3), thus avoiding the need for speculative instruction fetching. Hence, a BasicBlocker ISA needs at least one instruction that compares the operands and computes the target address. Operation 3) is handled implicitly by the bb instruction at the beginning of the basic block, which indicates after how many instructions the instruction pointer is updated to the target register. A BasicBlocker ISA *may* separate operation 1) and 2) arbitrarily. For example, an ARM version of BasicBlocker could keep the decoupled compare instruction. The branch instructions would only compute the target address based on the compare and the instruction pointer would be updated to the target address at the end of a basic block, indicated by the previous bb instruction.

## 4.6 Security

BasicBlocker was carefully designed with security in mind and the following section provides an overview of the security argument.

*4.6.1 Defense Against Spectre-type Attacks.* The first and foremost goal of BasicBlocker is to allow removing control-flow speculation to prevent Spectre-type attacks. CPUs that implement BasicBlocker should be designed after the following principle: *The microarchitectural state of a CPU is affected only by instructions that will eventually be retired.* Processors adhering to this principle are not allowed to do any type of control-flow speculation, including speculative fetching, as speculation always affects the microarchitectural state at least temporarily. This strict and simple design principle leads directly to the conclusion that the CPU is not vulnerable against any Spectre-type attack exploiting control-flow speculation, including *Spectre-PHT*, *Spectre-BPB*, and *Spectre-RSB* (taking the classification from Canella et al. [14]). BasicBlocker enables fast and efficient execution of code while maintaining the above stated principle.

Since BasicBlocker inherently does not provide mechanisms targeting the performance impact of disabling data-flow speculation (e.g. store-load forwarding, data cache prefetching), we consider attacks exploiting data-flow speculation such as *Spectre-STL* (again taking the classification from Canella et al. [14]) out of scope for this paper. It is the CPU designer's responsibility to prevent exploitation of data-flow speculation which can either be achieved by disabling it entirely or by implementing appropriate countermeasures. It is also possible to extend BasicBlocker to provide performance recovering mechanisms for data-flow speculation, e.g. by flagging allowed store to load forwarding code constructs at compile time, but we leave this for future work. We also do not discuss exception-based attacks such as Meltdown [40].

*4.6.2 Manipulation of BB Instruction Arguments.* In the following, we consider a powerful attacker that is able to manipulate the bb instruction arguments or the internal state of the bb registers. An attacker able to manipulate arguments of the bb instruction is in

control of certain parts of the control flow, by either flipping the sequential flag, decreasing the basic block size, or increasing the basic block size. Flipping the sequential flag will always lead to an exception, due to Definition 4.3. Decreasing the basic block size allows to skip the last instructions of a basic block, which might be critical, e.g. the removal of a secret key. Increasing the basic block size raises an exception in the *enforced BB* mode (Definition 4.4), but allows the execution of additional instructions in the legacy mode. Such additional instructions might be sufficient to form a covert channel, if the required gadgets can be found in the executable.

While those attacks may be harmful, this attacker model requires full control over the code executed on the victim's device and/or the register state. Generally, there are two points in time where an attacker can inject the manipulations described above: 1) at compile time and 2) at runtime. For 1), the attacker must be in control of the compiler which gives full control over the code anyway. In addition, a simple static analysis is sufficient to verify the correctness of all bb arguments of a specific binary. 2) Manipulation at run time comes down to either code injection or manipulation of internal values of the CPU for a particular program state, e.g. during a context switch or physical fault attack. Both, an attacker in control of the register state and an attacker able to perform code injection, have full control over the code executed by the victim's device in any case. BasicBlocker does not affect important OS security features like access rights management and therefore does not facilitate such attacks.

## 5 IMPLEMENTATION

We now give a specific example of BasicBlocker applied to an ISA, by defining BBRISC-V, a BasicBlocker modification of the RISC-V ISA. We further present a proof-of-concept implementation on a BBRISC-V soft core as well as a timing accurate simulator. To allow running a variety of benchmarks, we also provide a modified compiler for the BBRISC-V ISA.

Our modified ISA additionally specifies support for hardware loop counters, as proposed in Section 4.3, which we partly evaluate in Appendix A.

### 5.1 BBRISC-V ISA

The BasicBlocker modification requires the definition of the bb instruction as well as semantic changes to all control-flow instructions.

The bb instruction does not fit into any of the existing RISC-V instruction types so we defined a new instruction type to achieve an optimal utilization of the instruction bits. The instruction does not take any registers as input but rather parses the information directly from the bitstring. The size is encoded as a 16-bit immediate, enabling basic blocks with up to 65536 instructions. One can split a larger basic block into multiple sequential blocks if necessary. The sequential flag is a one-bit immediate value. The behavior of all RISC-V control-flow instructions (JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU) is changed so that they alter the control flow at the end of the current basic block.

### 5.2 CPU Implementation

*VexRiscv.* For the soft core variant of an in-order CPU, we chose the 32-bit VexRiscv core [43], written in SpinalHDL. This soft core is highly configurable by the use of plugins, which can be easily extended and modified to include new functionalities. We use a configuration with five stages (IF, ID, EX, MEM, WB) and 4096 byte, one-way instruction- and data caches. The result of control-flow instructions is available after the memory stage. We compare the modified BasicBlocker version of VexRiscv against the original core with the best available branch predictor (dynamic target). To enable a fair comparison, the BasicBlocker version has minimal configuration delta to the original core, that is we disabled control-flow speculation, by stalling the CPU until the next instruction is known, and added the logic described in Section 4. Other pipeline characteristics, e.g., cache misses behavior or jump target computation, remain unchanged.

Although speculation based attacks mostly get linked to out-of-order CPUs with deep pipelines, they are also feasible on smaller, in-order architectures that are more comparable to the VexRiscv [42].

*gem5.* To simulate the performance of CPUs with superscalar pipelines and out-of-order execution, we modified the 64-bit O3 CPU model of the gem5 simulator [8]. The gem5 implementation allows high configurability, for example arbitrary length pipelines can easily be simulated by modifying the delays between two stages.

In the default configuration, we use a superscalar pipeline with width 2, i.e. each pipeline stage can handle two instructions simultaneously. If not stated otherwise, we use the default configuration supplied in the *se.py* configuration file. The simulated CPU is equipped with 64kB L1 data cache and 32kB instruction cache. Using a 192 instruction entry sized reorder buffer, the CPU can execute instructions out-of-order. As for the VexRiscv implementation, the BasicBlocker version makes minimal configuration changes to enable a fair comparison of performance results.

### 5.3 Compiler Modification

To be able to evaluate the performance of our concept with well known benchmark programs we developed a compiler supporting and optimizing towards our instructions. Our compiler is based on the LLVM [37] Compiler Framework version 10.0.0, where we modified the RISC-V backend by introducing our ISA extension and inserting new compilation passes at the very end of the compilation pipeline to not interfere with other passes that do not support our new instructions.

First of all we split basic blocks for all occurrences of call instructions since they break the consecutive fetching and execution of instructions. As a next step we insert the bb instructions at the beginning of each basic block that include the number of instructions in the block. This is done directly before code emission to ensure that the number of instructions does not change due to optimizations. Linker relaxation, however, is one optimization that could reduce the number of instructions by substituting calls with a short jumping distance by a single jump instruction instead of two instructions (`aupic` and `jalr`). Since linker relaxation is not a major optimization, we simply disabled it, but it would also be possible to modify the linker to implement BasicBlocker-aware relaxation.

Our modifications to the semantics of terminating instructions (branches, calls, returns and jumps) allow them to be scheduled before the end of a basic block and rescheduling them earlier is also crucial to the performance of the code. This is done in a top-down list scheduler that is placed after register allocation and prioritizes terminating instructions. Additionally, we run another pass afterwards that relocates the terminating instructions to earlier positions in the basic blocks if this is supported by register dependencies.

## 6 EVALUATION

In the following we provide a performance evaluation of BasicBlocker on VexRiscv and gem5 by comparing the execution time of different variants of the two CPUs. Thereby, special care is given to the impact of CPU features and code characteristics.

### 6.1 Selection of Benchmarks

Both implementations of BasicBlocker presented in this paper enforce the presence of exactly one bb instructions in every basic block (i.e. misplaced or missing bb instructions cause a program to crash). This ensures that the benchmarks only measure the performance of BasicBlocker without noise from legacy code snippets, e.g. library functions, but also requires all code to be compiled by our modified compiler. Since this forces us to perform the benchmarks bare-metal (i.e. without OS support), it is quite difficult to run typical user level benchmarks such as SPEC.

We chose the benchmarks included in the Embench benchmark suite [25], the well-known *Coremark* benchmark [26] and our own pointer-chasing benchmark for our evaluation. The selection of programs within the Embench suite resemble code from different use cases such as cryptography (*nettle-sha*, *nettle-aes*), image processing (*picojpeg*) and matrix multiplication (*matmult-int*). For three of the programs we also included our own optimized version (*-opt*), targeted at general architectures and discussed in more detail in Appendix B. All those programs are characterized by minimal dependencies and are thus well suited for bare-metal benchmarking.

Since all of the benchmarks require the `libc` library (and some also `libm`), we compiled Newlib [30] using our modified LLVM compiler. However, some of the benchmark programs require further dependencies, e.g. `libgcc`, and could thus not be compiled for our target. For the evaluation we included all available benchmark programs that compiled with the modified `libc` and `libm` and passed the test for functional correctness.

We compiled three versions of each benchmark program, as listed in Table 1: one without BasicBlocker, one with a new compile flag enabling the insertion of bb instructions, and one with bb plus rescheduling of terminator instructions. Except for these differences, the compiler and compile flags are identical. The compile flags are listed in Appendix C. We ran those programs on several variants of VexRiscv and gem5, as listed in Table 2. The simplest non-speculative variant (*NoSpec*) disables branch prediction and speculative fetching. The control-flow speculation configuration (*CFS*) implements the unmodified version of the CPU with the default branch predictor. As we execute our benchmarks bare-metal, we observe only minimal noise through the microarchitectural state of the VexRiscv. The gem5 platform has no noise at all, as it is a deterministic simulation with a reset prior to each run.

**Table 1: Compiled versions used for benchmarking.**

| Name | Description |
| --- | --- |
| *Baseline* | Standard RISC-V version. |
| *BB Info* | As in Baseline, but every basic block starts with a bb instruction. |
| *BB Resched* | As in BB Info, but with high-priority rescheduling of terminator instructions. |

**Table 2: Processor instantiation options. BB: supports** bb **instruction. SF: speculative fetching. BP: branch predictor**

| Name | BB | SF | BP |
| --- | --- | --- | --- |
| NoSpec | no | no | no |
| Control-Flow Speculation (CFS) | no | yes | yes |
| *BasicBlocker* (this work) | yes | no | no |

### 6.2 VexRiscv Evaluation

We first evaluate the performance of BasicBlocker on VexRiscv, which resembles a small-scale, in-order, embedded-like processor, by comparing the execution time of the CPU variants in Table 2 together with the program versions of Table 1. We chose the strictly non-control-flow-speculative processor as a naive but secure baseline and report the relative execution time of the other variants in Figure 4. The average speedup over all benchmarks is 2.88× and 2.12× for the version using control-flow speculation (*CFS*) and the BasicBlocker version with instruction rescheduling (*BB Resched*), respectively. The maximal and minimal speedups are 3.93× (*crc32*) and 1.44× (*pointer-chase*) for control flow speculation and 3.09× (*crc32-opt*) and 1.07× (*pointer-chase*) for BasicBlocker with rescheduling.

For several benchmarks the speedup of control-flow speculation is comparable to BasicBlocker with instruction rescheduling. This is true for *ud*, *matmult-int*, *nettle-sha*, *nettle-aes*, and *crc32-opt*. For *nettle-aes* and *crc32-opt* BasicBlocker with instruction rescheduling even outperforms control-flow speculation (speedup of 2.88× vs. 2.78× and 3.09× vs. 2.79× respectively). This is possible as with enough rescheduling opportunities no pipeline stalls are necessary at all. For other benchmarks, control-flow speculation outperforms BasicBlocker with a larger margin (e.g. *minver*, *Coremark*, *nbody*, and *huffbench*).

In general, BasicBlocker performs best for benchmarks that have large basic blocks and less branches (e.g. *nettle-aes*, and *nettle-sha*) whereas the large difference of speedup between control-flow speculation and BasicBlocker occurs for branch heavy code with small basic blocks (e.g. *minver*). A more thorough analysis of code characteristics is given in Section 6.4. We emphasize that many optimization techniques for execution time tend also to prefer large basic blocks with less branches over small basic blocks with a lot of branches, e.g. loop unrolling, or function inlining.

### 6.3 Gem5 Evaluation

We conduct the same performance analysis with the gem5 simulator, which resembles a more sophisticated, out-of-order, and multi-scalar processor. Again, the strictly non-control-flow-speculative
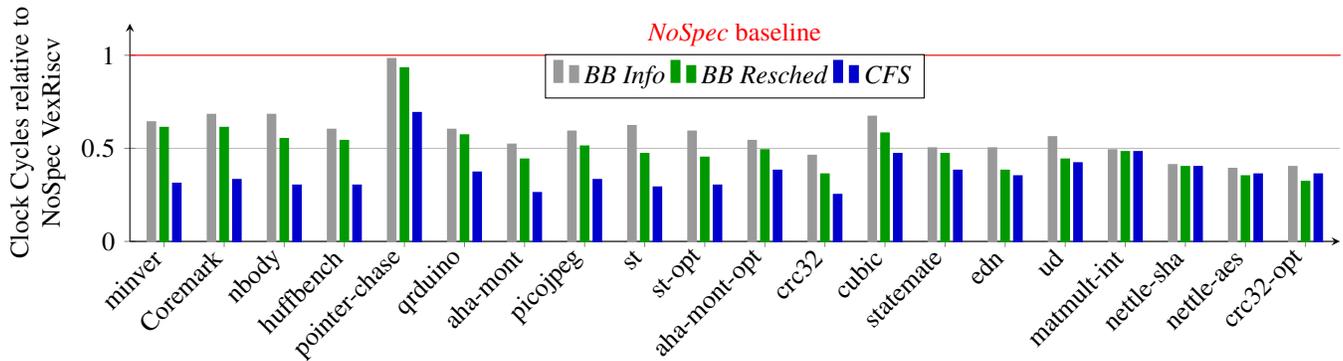
**Figure 4: Performance results for various benchmarks on VexRiscv measured in clock cycles. The results are relative to the *NoSpec* configuration of VexRiscv (red line). Sorted descended by speedup delta in *BB Resched* vs *CFS* case. Lower delta is better. For abbreviations see Tables 1 and 2.**
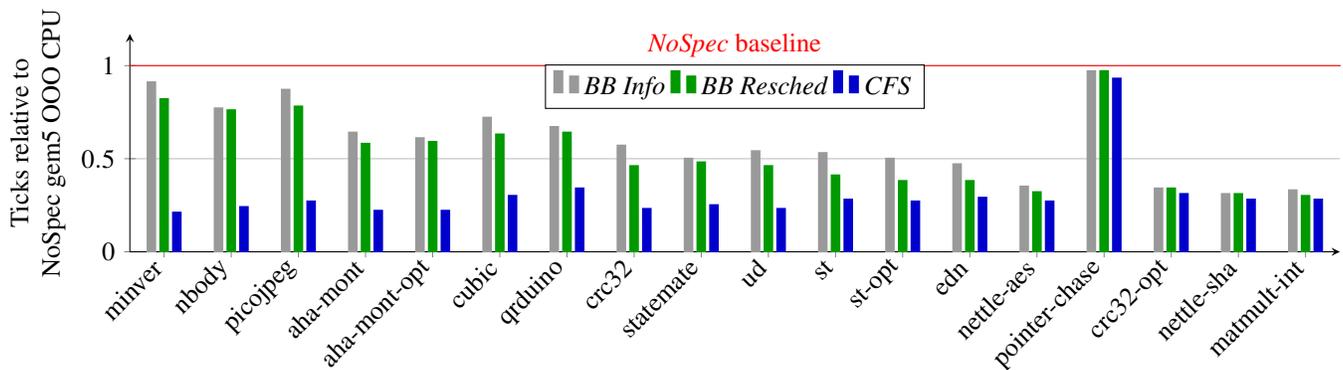


**Figure 5: Performance results for various benchmarks on gem5 measured in simulation ticks. The results are relative to the *NoSpec* configuration of gem5 (red line). Sorted descended by speedup delta in *BB Resched* vs *CFS* case. Lower delta is better. For abbreviations see Tables 1 and 2. *Huffbench* and *Coremark* did not compile for the 64-bit target.**

processor variant serves as a naive but secure baseline. The gem5 CPU model processes up to two instructions in every clock cycle. The strictly non-speculative version cannot utilize this capacity as fetching multiple instructions at once implies speculative fetching. The relative execution time of the benchmarks for the evaluated processor variants are reported in Figure 5. The average speedup over all running benchmarks is 3.69× and 2.13× for the version using control-flow speculation and BasicBlocker with rescheduling of instruction respectively. The maximum and minimum speedups are 4.80× (*minver*) and 1.07× (*pointer-chase*) for control-flow speculation and 3.09× (*crc32-opt*) and 1.07× (*pointer-chase*) for BasicBlocker with rescheduling. Hence, the speedup achieved by BasicBlocker on gem5 is overall comparable to the speedup achieved on VexRiscv and for well performing cases slightly higher. However, the speedup achieved by the means of control-flow speculation is higher than in the VexRiscv example.

Taking a closer look at specific benchmarks reveals again some cases where BasicBlocker matches the performance of control-flow speculation, e.g. *pointer-chase crc32-opt*, *nettle-sha*, or *matmult-int* while for others control-flow speculation is considerably faster,

e.g. *minver*, *nbody*, or *picojpeg*. As analyzed in the following, the code characteristics have a high influence on the performance. The low speedup for *pointer-chase* at all gem5 architectures is expected, as memory-access time clearly dominates any pipeline characteristic for this benchmark. The results show the applicability of BasicBlocker on superscalar, out-of-order processors. We further analyze the influence of processor characteristics in Section 6.5.

## 6.4 Influence of Code Characteristics

To analyze how the structure of the code influences the performance of BasicBlocker, we evaluate the code characteristics of each benchmark regarding the average size of basic blocks and average rescheduling of control-flow instructions. Since the impact of basic blocks that are executed frequently during the benchmarks is higher than those that are executed only once, we perform a dynamic hotspot analysis and weight the results based on the frequency of invocation. In Figure 6 the resulting distribution of basic block sizes is pictured. The figure shows, that there are strong differences in the basic block sizes for the benchmarks. For *matmult-int*, *nettle-aes* and *nettle-sha*, the highest arithmetic average size of the
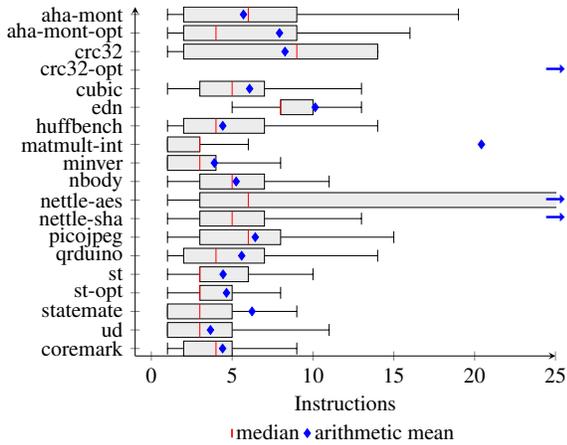
Figure 6: Distribution of basic block sizes, weighted by the number of invocations, dynamically derived from the hotspot analysis.
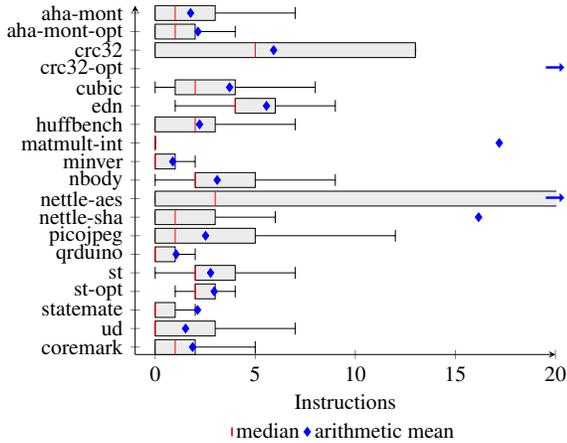


Figure 7: Distribution of instruction rescheduling per basic block, weighted by the number of invocations, dynamically derived from the hotspot analysis.

basic blocks executed during the benchmark is reached with more than 25 instructions, whereas *minver* and *coremark* have a relatively small average basic block size, below five instructions. The optimized versions of *aha-mont*, *crc32* and *st* increase the mean basic block size by enabling more inlining and thus contribute to a smaller delta in the benchmarks between the BasicBlocker and speculative version of the CPU. For *crc32-opt* the distribution of basic block sizes changed dramatically and lead to a speedup of 2.13× and more for all CPU versions compared to the original benchmark.

Figure 7 shows the average number of instructions that follow the control flow instruction (this is only relevant for the *BB Resched* case, not for the *BB Info*). The intuitive assumption is that large basic blocks allow for higher rescheduling of control flow instructions. This assumption is confirmed by the results shown in the figure. While the average rescheduling number for the aforementioned
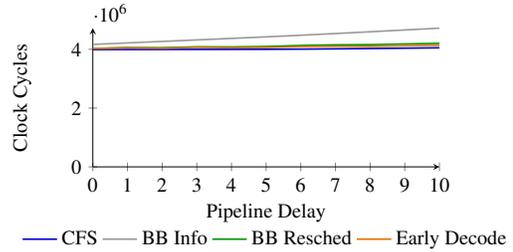


Figure 8: Influence of additional pipeline stages on the execution time for the benchmark *matmult-int* on VexRiscv.
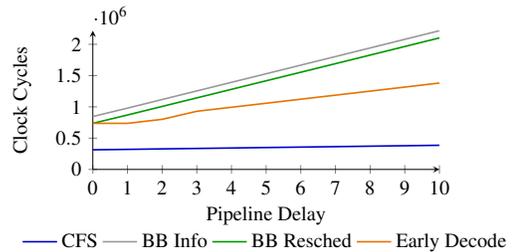


Figure 9: Influence of additional pipeline stages on the execution time for the benchmark *minver* on VexRiscv.

benchmarks with large basic blocks is high (above 15 instructions on average), benchmarks with smaller basic blocks such as *Coremark* and *minver* offer less average rescheduling opportunities.

The performance results in Figure 4 and 5 show, that programs with large basic blocks in their core functions (and therefore good rescheduling opportunities) perform better with BasicBlocker than those benchmarks with small basic blocks. For real world workloads, the core functions that are regularly executed are often well optimized and - in many cases - try to avoid branches to gain improved performance [16, 21, 22].

## 6.5 Influence of Pipeline Characteristics

*Pipeline Length.* We analyze the influence of additional pipeline stages on the execution time of our benchmarks to give an estimation of run time on other CPU architectures. As for space restrictions we analyze the influence of the pipeline length for a smaller sample of the above shown benchmarks. With *matmult-int* and *minver*, we chose one well performing benchmark and one with higher performance penalty. We modified the VexRiscv soft core and placed additional dummy pipeline stages between fetch and decode such that the original architecture has a pipeline delay of zero and each additional stage increments the pipeline delay by one. The results are shown in Figure 8 and Figure 9 for *matmult-int* and *minver* respectively.

The data clearly show that additional pipeline stages have nearly no effect when control-flow speculation is used (CFS), which is expected as the longer pipeline only introduces a penalty if a missprediction occurs. Also, the linearly increasing penalty for the naive BasicBlocker implementation is to be expected, since a constant amount of additional clock cycles is added to all transitions
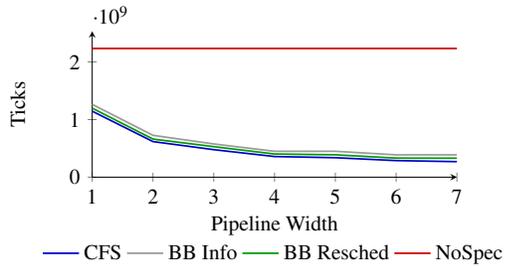
**Figure 10: Influence of superscalarity on the performance of BasicBlocker using the *matmult-int* benchmark on gem5.**



**Figure 11: Influence of superscalarity on the performance of BasicBlocker using the *minver* benchmark on gem5.**

between basic blocks. More interesting is the case where the compiler is allowed to reschedule control-flow instructions. Here we can see clear differences between the benchmarks. While the impact of additional stages is only small and non-linear in the case of *matmult-int* running on VexRiscv, we can observe a mirroring of the naive BasicBlocker behavior for *minver* running on VexRiscv. We can explain this as an artifact of the code structure, as discussed earlier. *Minver* is composed of mostly small basic blocks resulting in only a few rescheduling options. Hence, the impact of the longer pipeline is preserved nearly entirely. In contrast, *matmult-int* has better options for rescheduling, and hence, the penalty can be better absorbed through the early determination of the next basic block.

We also analyzed one additional configuration, where we implemented a decoding of the bb instruction directly after the instruction cache, and hence, before the pipeline delay is introduced. Figures 8 and 9 show that this can reduce the performance impact of longer pipelines, as the penalty only occurs for the computation of the next basic block and not for the determination of the basic block length and sequential flag.

We conducted a similar analysis for the gem5 out-of-order processor and the results show the same behavior as the discussed examples.

*Superscalarity.* By using superscalarity modern processors can process several instructions in parallel within a single clock cycle. We, therefore, modify our gem5 implementation to evaluate the performance impact of superscalar processors using BasicBlocker. As described above, our default configuration for the gem5 uses a 2× superscalar pipeline. Figure 10 and 11 show the performance results for an up to 7× superscalar pipeline for *matmult-int* and *minver* respectively.

The red line in Figures 10 and 11 show the strictly non-speculative version of the CPU. Since it is not allowed to do speculative fetching, only one instruction can be fetched at a time. Thus, the superscalarity has no effect in this scenario. The results for the well-performing *matmult-int* benchmark strikingly demonstrate the potential of BasicBlocker using superscalar pipelines. The *bb info* version as well as the *rescheduled* version incur minimal performance overhead over the original configuration using speculation. That is, large basic blocks allow optimal utilization of the superscalar pipeline. For the *minver* benchmark, which has much smaller basic blocks, it shows that the additional pipeline slots can barely be filled for a
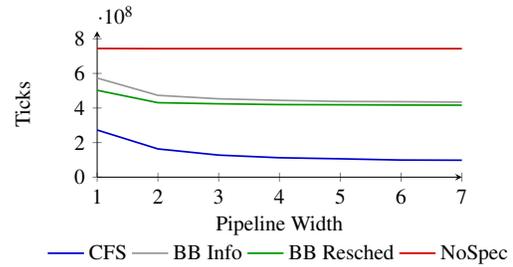
superscalarity larger than two. The lines for *bb info* and *rescheduled* converge for a large pipeline width. That is, small basic blocks will eventually be fetched within a single clock cycle, making any rescheduling irrelevant to the performance.

## 7 CONCLUSION

In this work, we demonstrated a universal countermeasure against control-flow speculation attacks such as Spectre. We have chosen a path of conservative security assumptions that completely address a large number of current and upcoming attacks. Despite the consensus that control-flow speculation is inevitable, we demonstrate in this work a practical alternative.

We propose a novel concept to transport control-flow information from the software to the hardware, enabling practical implementations of strictly non-control-flow-speculative processors. The performance evaluation clearly shows that BasicBlocker maintains current levels of performance for code with large basic blocks, a characteristic that is common in highly optimized code (i.e. function inlining, loop unrolling). For branch-heavy code control-flow speculation is clearly faster, however, this is at the cost of security.

In contrast to other work, BasicBlocker allows to remove control-flow speculation, including speculative-fetching, entirely, and hence, tackles a basic necessity for a complete class of attacks. This simplifies the security analysis, is securely backwards compatible, and the resulting code is independent of the underlying microarchitecture.

We showcase our concept by specifying the BBRISC-V ISA, including a concrete implementation of that ISA based on VexRiscv and gem5, accompanied by an optimizing compiler that rests on the LLVM Compiler Framework. We emphasize that BasicBlocker is a generic solution that can be applied to other ISAs as well. Our prototype implementations show that BasicBlocker is applicable for a variety of processor types and we point to code-optimization strategies, that can further enhance the performance.

In Section 4.3 we present possible architectural and microarchitectural improvements for BasicBlocker that are interesting to investigate in future work. More aggressive inlining and linearization within superblocks are opportunities to reduce the amount of control-flow instructions emitted by the compiler. By taking the algorithmic level into consideration further optimizations can be achieved; see Appendix B. In addition, we expect extensions to improve the performance and security of BasicBlocker, most notably hardware loop counters, that can be seamlessly integrated into our

concept (see Appendix A), or extensions dealing with fault-based transient-execution attacks.

## ACKNOWLEDGMENTS

## REFERENCES

[1] 2020. *Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.* Technical Report. ARM.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[3] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings (Lecture Notes in Computer Science)*, Stefan Mangard and François-Xavier Standaert (Eds.), Vol. 6225. Springer, Santa Barbara, CA, 110–124. https://doi.org/10.1007/978-3-642-15031-9_8

[4] Krste Asanović and David A. Patterson. 2014. Instruction Sets Should Be Free: The Case For RISC-V. (2014). https://people.eecs.berkeley.edu/~krste/papers/EECS-2014-146.pdf.

[5] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. Specshield: Shielding speculative data from microarchitectural covert channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 151–164.

[6] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos V. Rozas, Adam Morrison, Frank McKeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa R. Alameldeen. 2020. Speculative Interference Attacks: Breaking Invisible Speculation Schemes. *CoRR* abs/2007.11818 (2020). arXiv:2007.11818 https://arxiv.org/abs/2007.11818

[7] Daniel J. Bernstein. 2019. djbsort. (2019). https://sorting.cr.yp.to/

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[9] Erich Bloch. 1959. The engineering design of the Stretch computer. In *Papers presented at the December 1–3, 1959, eastern joint IRE-AIEE-ACM computer conference.* 48–58.

[10] Benjamin A Braun, Suman Jana, and Dan Boneh. 2015. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189* (2015).

[11] Brad Calder and Dirk Grunwald. 1994. Fast and Accurate Instruction Fetch and Branch Prediction. In *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*, David A. Patterson (Ed.). IEEE Computer Society, 2–11. https://doi.org/10.1109/ISCA.1994.288166

[12] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. 2019. Fallout: Leaking data on meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 769–784.

[13] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N Khasawneh. 2020. Evolution of defenses against transient-execution attacks. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI.* 169–174.

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium.* extended classification tree at https://transient.fail/.

[15] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[16] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. 2001. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34.* Citeseer, 182–182.

[17] Douglas W. Clark and Henry M. Levy. 1982. Measurement and analysis of instruction use in the VAX-11/780. (1982), 9–17 pages. https://dl.acm.org/doi/pdf/10.1145/1067649.801709.

[18] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC).* IEEE, 1–6.

[19] John A. DeRosa and Henry M. Levy. 1987. An Evaluation of Branch Architectures. In *Proceedings of the 14th Annual International Symposium on Computer Architecture. Pittsburgh, PA, USA, June 1987*, Daniel C. St. Clair (Ed.). 10–16. https://doi.org/10.1145/30350.30352

[20] Scott DiPasquale, Khaled Elmeleegy, CJ Ganier, and Erik Swanson. 2003. Hardware Loop Buffering. (2003).

[21] Reem Elkhouly, Ahmed El-Mahdy, and Amr Elmasry. 2015. Pattern-Driven Branchless Code Generation. *JEC-ECC* (2015).

[22] Amr Elmasry and Jyrki Katajainen. 2013. Branchless search programs. In *International Symposium on Experimental Algorithms*. Springer, 127–138.

[23] Joseph A Fisher. 1983. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th annual international symposium on Computer architecture.* 140–150.

[24] Agner Fog. 2020. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. (2020). https://www.agner.org/optimize/.

[25] Free and Open Source Silicon Foundation. 2020. Embench IOT. https://www.embench.org/. (May 2020). Accessed: 2020-05-29.

[26] Shay Gal-On and Markus Levy. 2012. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium* (2012).

[27] Shay Gueron. 2010. Intel Advanced Encryption Standard (AES) New Instructions Set. (2010). https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[28] Linley Gwennap. 2010. Sandy Bridge spans generations. (2010). http://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/Microprocessor-Report-Sandy-Bridge-Spans-Generations-243901.pdf.

[29] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass. (2018). https://bugs.chromium.org/p/project-zero/issues/detail?id=1528/.

[30] J Johnston and T Fitzsimmons. 2021. The newlib homepage. *URL http://sourceware.org/newlib* (2021).

[31] Toni Juan, Sanji Sanjeevan, and Juan J. Navarro. 1998. Dynamic History-length Fitting: A Third Level of Adaptivity for Branch Prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, ISCA 1998, Barcelona, Spain, June 27 - July 1, 1998*, Mateo Valero, Gurindar S. Sohi, and Doug DeGroot (Eds.). IEEE Computer Society, 155–166. https://doi.org/10.1109/ISCA.1998.694771

[32] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. 2018. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *CoRR* abs/1806.05179 (2018).

[33] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.

[34] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[36] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18).*

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[38] Edward A. Lee, Jan Reineke, and Michael Zimmer. 2017. Abstract PRET Machines. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017.* 1–11. https://doi.org/10.1109/RTSS.2017.00041

[39] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 264–276.

[40] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security*

*Symposium (USENIX Security 18).* 973–990.

[41] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 2109–2122.

[42] Hamed Nemati, Roberto Guanciale, Pablo Buiras, and Andreas Lindner. 2020. Speculative Leakage in ARM Cortex-A53. *arXiv preprint arXiv:2007.06865* (2020).

[43] Charles Papon. 2020. VexRiscv. https://github.com/SpinalHDL/VexRiscv. (May 2020). Accessed: 2020-05-28.

[44] Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Francky Catthoor, and Diederik Verkest. 2008. Distributed loop controller for multithreading in unithreaded ILP architectures. *IEEE Trans. Comput.* 58, 3 (2008), 311–321.

[45] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient invisible speculative execution through selective delay and value prediction. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 723–735.

[46] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. Context: A generic approach for mitigating spectre. In *Proc. Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss*, Vol. 10.

[47] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 753–768.

[48] Jakub Szefer. 2019. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security* 3, 3 (2019), 219–234.

[49] Andrew S Tanenbaum. 2016. *Structured computer organization.* Pearson Education India.

[50] Jan Philipp Thoma, Christian Niesler, Dominic A. Funke, Gregor Leander, Pierre Mayr, Nils Pohl, Lucas Davi, and Tim Güneysu. 2021. ClepsydraCache - Preventing Cache Attacks with Time-Based Evictions. *CoRR* abs/2104.11469 (2021). arXiv:2104.11469 https://arxiv.org/abs/2104.11469

[51] Paul Turner. 2018. Retpoline: A software construct for preventing branch-target-injection. *URL https://support. google. com/faqs/answer/7625886* (2018).

[52] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18).* 991–1008.

[53] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. Addendum to RIDL: Rogue in-flight data load. (2019). https://mdsattacks.com/.

[54] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP).* IEEE, 88–105.

[55] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. 2014. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symposium (USENIX Security 14).* 687–702.

[56] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 572–586.

[57] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018). See also USENIX Security paper Foreshadow.

[58] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[59] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. 2020. Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core. *arXiv preprint arXiv:2005.02193* (2020).

[60] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 428–441.

[61] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing.. In *NDSS.*

[62] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52).* Association for Computing Machinery, New York, NY, USA, 954–968.

https://doi.org/10.1145/3352460.3358274

[63] Drew Zagieboylo, G Edward Suh, and Andrew C Myers. 2019. Using information flow to design an isa that controls timing channels. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF).* IEEE, 272–27215.

[64] Yinqian Zhang and Michael K Reiter. 2013. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* 827–838.

[65] Lutan Zhao, Peinan Li, Rui Hou, Jiazhen Li, Michael C Huang, Lixin Zhang, Xuehai Qian, and Dan Meng. 2020. A Lightweight Isolation Mechanism for Secure Branch Predictors. *arXiv preprint arXiv:2005.08183* (2020).

# A  LOOP COUNTER

Loops are often the execution hotspots in programs and contribute considerably to diverging control flow. Therefore the concept of hardware supported loops can be profitable as already discussed in the literature [20, 44] and implemented in various architectures.

In general, hardware loop counters are realized by a hardware counter which is set by a dedicated instruction with a value representing the maximum trip count for the loop. The trip count must be computable at compile time to be inserted by an immediate value or available in a register at run-time before entering the loop. Information about which instructions are included in the loop is expressed via labels or additional specific instructions. The hardware loop counter decrements the start value after each iteration and induces a branch back to the start of the loop as long as the counter is unequal to zero. This can be done implicitly at the end of the loop or explicitly with an instruction.

Performance improvements by the usage of hardware loops result from reduced instruction size and dedicated loop control logic that does not have to be calculated by the ALU. For our BasicBlocker concept, hardware loops are actually much more valuable for performance when only applied to loops that will not terminate early, because in this case the control flow for all loop iterations is known when entering the loop.

We can seamlessly support hardware loop counters in our design concept, by introducing a new instruction and adding two arguments to the bb instruction. The new I-Type instruction named `lcnt` requires a 12 bit immediate value as well as a source and a target register. The counter value is then computed as $cnt = imm + rs.value$ and saved to the loop counter set defined in $rd$. To fully support loop counters we also add four start and end flags to the bb instructions, to support a maximum of four loop counter sets. The two flags in the bb instruction are necessary for each loop counter set, which means that the bb instruction needs $2n$ bits to support $n$ loop counter sets.

```
bb 2, 1, 00, 00 ; len = 2, seq = 1          1
add a0, a0, a1                              2
lcnt 3, lc1 ; 3 iterations, set 1           3
bb 2, 0, 01, 01 ; loop start/end            4
add a1, a2, a2                              5
mul a2, a1, a2                              6
bb 7, 0, 00, 00 ; after loop                7
```

**Listing 1: Single basic block loop with 3 iterations in counter set 1; Colors correspond to the execution trace in 2**

Listing 1 shows the exemplary use of the hardware loop counter. In line 3, the counter in loop set `ls1` is initialized to 3. The following

bb instruction has the start- and end flag for loop set 1 enabled which indicates a loop that starts at the beginning of this basic block and stretches until the end of the same basic block. Each bit in the flags represents one loop counter set, allowing nested loops with the same start- or end address and nested loops sharing the same basic block as start or end. It is possible to model loops that stretch across multiple basic blocks by setting the start and end flags in the respective basic blocks accordingly. When the bb instruction with the start flag is executed, the current *PC* is saved as start address in the corresponding loop counter set. Simultaneously, the counter value of that set is decremented by one. When the execution reaches the bb instruction with the corresponding end flag, the target address (which determines where the CPU continues execution) is set to the corresponding start address if the counter is not zero. Otherwise, the basic block is handled like a normal sequential basic block and the loop will exit.

```
bb 2, 1, 00, 00 ; len = 2, seq. block
bb 2, 0, 01, 01 ; loop: start L1, end L1
add a0, a0, a1
lcnt 2, lc1 ; 2 iterations, set 1
bb 2, 0, 01, 01 ; loop: start L1, end L1
add a1, a2, a2
mul a2, a1, a2
bb 2, 0, 01, 01 ; loop: start L1, end L1
add a1, a2, a2
mul a2, a1, a2
bb 7, 0, 00, 00 ; after loop
add a1, a2, a2
mul a2, a1, a2
```

**Listing 2: Execution trace of CPU with color matched instructions to the code sequence in 1.**

In Listing 2, the instruction trace of the program snippet from Listing 1 is shown as it is executed by the CPU. Since the first bb instruction indicates a sequential basic block, the CPU immediately fetches the bb instruction of the next basic block which notifies the fetch unit that the second basic block is the start and end block of the loop. After that, the remaining add and lcnt instructions are executed to finish the first basic block. From now on the loop counter determines the execution flow. Since the second basic block is the only basic block of the loop, the bb instruction of this block is fetched again, to prepare the second loop round, before the basic block is executed to complete the first round. This happens again until the loop counter is zero, resulting in fetching the last bb instruction, to exit the loop, before the last round of the loop is executed. Afterwards the execution continues outside of the loop with the normal instruction flow.

We Implemented our proposed hardware loop counter concept in the VexRiscv core and added elementary compiler support for one loop counter set. Because the loop counter can only be used for loops that do not contain calls and have a fixed trip count, it can only be applied by the compiler to a small subset of the loops in the benchmarks. While the impact of the hardware loop counter is neglegtable for most benchmarks, it substantially improves the

speed on others. The speedup for *edn* improved from 2.63× to 2.70×, getting closer to the 2.85× speedup of the speculative version compared to the non-speculative baseline. For *ud* the hardware loop counter enabled the BasicBlocker variant to match the speed of the speculative version. The biggest impact can be observed for *aha-mont* where the speedup increased from 2.27× to 3.13×.

## B SYNERGIES BETWEEN BASICBLOCKER AND ALGORITHMIC IMPROVEMENTS

There are continual announcements of performance improvements in software packages to handle computational "hot spots", such as the inner loops in audio/video processing. The main point of this appendix is that the natural pursuit of higher-speed software favors BasicBlocker: software changes that improve performance on current non-BasicBlocker CPUs tend to produce even larger improvements on BasicBlocker CPUs.

### B.1 Dimensions of performance analysis

The performance evaluation in Section 6 focuses on measuring the impact of changing (1) an existing CPU with an existing compiler to (2) a BasicBlocker CPU with a BasicBlocker-aware compiler. Each of the benchmarks being compiled and run—for example, the st software in the middle of the graphs in that section—is treated as being set in stone. There is no effort in Section 6 to modify st for better performance, whether by explicit changes in the st code or by additions to the compiler's built-in optimizations beyond the BasicBlocker support described earlier.

This appendix instead treats the software as a third variable beyond the compiler and the CPU, reflecting the reality that software evolves for the pursuit of performance. For example, we modified the st software to obtain the st-opt software described below, computing the same results as st at higher speed. Our goal in changing st to st-opt was to match what typical programmers familiar with performance would naturally do if st turned out to be a bottleneck. We used a profiler (specifically gcc -pg) to see bottlenecks on an existing CPU (specifically the ARM Cortex-A7 CPU in a Raspberry Pi 2), inspected the software to identify underlying inefficiencies, and removed those inefficiencies, while retaining portability.

We selected three case studies for these software modifications: st, aha-mont, and crc32. We were aiming here for a spread of different types of code. Within our benchmarks, aha-mont is at the worst quartile for BasicBlocker, while st and crc32 are slightly better than median; st uses floating-point arithmetic, while aha-mont and crc32 do not.

It is important to observe that our modifications remove *cross-platform* inefficiencies. Switching from st, aha-mont, and crc32 to st-opt, aha-mont-opt, and crc32-opt saves time on current CPUs. The same changes save time on BasicBlocker—and, as our measurements show, reduces the cost of BasicBlocker compared to current CPUs. We summarize the inefficiencies below for each case study, and explain why the benefits for BasicBlocker should not be viewed as a surprise.

### B.2 From st to st-opt

See the full version of this paper. Brief summary: Embench describes st as a "statistics" benchmark. The benchmark computes

statistics regarding two arrays. However, profiling immediately shows that most of the time in st is spent initializing the arrays. There are many easily removable inefficiencies in st, such as redundant loops recomputing variances, a loop constantly calling a separate RandomInteger function, and failures of unrolling. The overall increase from st compiled code size to st-opt compiled code size is negligible: around 100 bytes, depending on the instruction set.

### B.3 From aha-mont to aha-mont-opt

See the full version of this paper. Brief summary: Embench describes aha-mont as a "Montgomery multiplication" benchmark. Montgomery multiplication is a well-known method to carry out integer operations modulo a specified odd modulus $m$ without using divisions by $m$. The aha-mont code is a slightly modified version of a snippet from Warren's "Hacker's Delight" code corpus. However, profiling shows that 65% of the aha-mont time is spent in divisions by $m$; Warren's snippet includes a main routine with tests, and the tests use divisions. We took faster division code from Warren's corpus and incorporated it into aha-mont-opt.

### B.4 From crc32 to crc32-opt

See the full version of this paper. Brief summary: Embench describes crc32 as a "CRC error checking 32b" benchmark. Profiling once again shows that most of the time is spent on something else: each iteration of the crc32 loop calls a function *in a separate file* to generate random input. The compiler does not inline the function. The only changes from crc32 to crc32-opt are (1) putting the random-number-generation function into the same file for inlining and (2) marking the main loop with UNROLL(4). The unrolling increases code size, while the inlining reduces code size since unnecessary function prologs and epilogs disappear; both changes in code size are negligible.

### B.5 Patterns observed, and consequences for BasicBlocker

In each of these case studies, many of the inefficiencies in the original code arise directly from loop overhead (and, analogously, function-call overhead in the crc32 case). Branch prediction does not magically make loop overhead (and function-call overhead) disappear; it can reduce the overhead, but extremely short loops (and functions) are generally performance problems if they are in hot spots. The standard response is unrolling (plus inlining) for hot spots, saving time on current CPUs—and saving even more time for BasicBlocker.

Further inefficiencies were handled by copy elimination (e.g., removing the repeated reads and writes of *Sum), strength reduction (e.g., replacing divisions by 8095 with multiplications), and common-subexpression elimination (e.g., eliminating the repeated computation of variance)—which can indirectly *increase* branch frequency by reducing the time spent on arithmetic operations between branches. However, having fewer instructions in a loop usually allows more unrolling for the same code size, and then branch frequency drops again.

BasicBlocker avoids all hot-spot stalls if each hot-spot branch condition can be computed enough cycles ahead of the branch to cover the pipeline length. The obvious way to find computations that are *intrinsically* bad for BasicBlocker, rather than being bad as a result of easily fixable failures of unrolling and inlining, is to look for computations bottlenecked by one data-dependent branch feeding into another data-dependent branch, such as the bit-by-bit data-dependent branches in modul64 and xbinGCD inside aha-mont. We emphasize that these computations also perform poorly on existing CPUs; we saved time across platforms by replacing these algorithms with faster algorithms.

These case studies are not necessarily representative. Are there important computations where the *fastest* algorithms involve one data-dependent branch after another? There is a textbook example at this point, namely sorting integer arrays. Embench includes a wikisort benchmark (which did not compile for our target), stably sorting 400 64-bit records, where each record has a 32-bit integer key used for sorting and 32 bits of further data. The algorithm used inside wikisort is a complicated merge-sort variant; overall wikisort has 1117 lines, several kilobytes of compiled code.

However, the textbook picture of the fastest sorting algorithms has been challenged by the recent speed records in [7] for sorting various types of arrays on Intel CPUs. The software in [7] has no data-dependent branches. For a size-400 array, this software uses a completely predictable pattern of 7199 comparators (size-2 sorting operations, i.e., min-max operations); merge sort, heap sort, etc. use half as many comparisons but in an unpredictable pattern, incurring so much overhead as to be non-competitive.

This raises a research question: exactly how far is wikisort from optimal on smaller CPUs? An application where sorting is critical will select the fastest sorting routine from among many options—not just comparison-based sorts such as merge sort but also radix sort, sorting networks, etc. The time taken by wikisort is, presumably, an overestimate of the time needed for the same task on current CPUs, and an even more severe overestimate of the time needed for the same task on BasicBlocker CPUs.

More broadly, algorithms without data-dependent branches are an essential part of the modern software-optimization picture for large CPUs, especially because of the role of these algorithms inside vectorized code. This does not imply that these algorithms have the same importance on today's smaller CPUs, but in any case they are among the options available for small and large BasicBlocker CPUs. Taking advantage of this software flexibility brings BasicBlocker CPUs even closer to current CPUs in overall performance.

## C COMPILE FLAGS

This Appendix lists the flags used to compile the benchmarks. For Coremark, we used O3, march = rv32im, mabi = ilp32, target = riscv32-unknown-elf, mno-relax, lc, nostartfiles, ffree-standing and for Embench we used O3, march = [rv32im / rv64imfd], mabi = [ilp32/lp64d], riscv[32/64]-unknown-elf, mno-relax, fnostrict-aliasing.