Saving Energy with Per-Variable Bitwidth Speculation

Tommy McMichen*
Northwestern University

David Dlott*
Northwestern University

Panitan Wongse-ammat
Northwestern University

Nathan Greiner
Northwestern University

Hussain Khajanchi
Northwestern University

Russ Joseph
Northwestern University

Simone Campanoni
Northwestern University

Abstract

Tiny devices have become ubiquitous in people's daily lives. Their applications dictate tight energy budgets, but also require reasonable performance to meet user expectations. To this end, the hardware of tiny devices has been highly optimized, making further optimizations difficult. In this work, we identify a missed opportunity: the bitwidth selection of program variables. Today's compilers directly translate the bitwidth specified in the source code to the binary. However, we observe that most variables do not utilize the full bitwidth specified in the source code for the *majority* of execution. To leverage this opportunity, we propose BitSpec: a system that performs fine-grained speculation on the bitwidth of program variables. BitSpec is implemented as a compiler-architecture co-design, where the compiler transparently reduces the bitwidth of program variables to their *expected* needs and the hardware monitors speculative variables, reporting misspeculation to the software, which re-executes at the original bitwidth, ensuring correctness. BitSpec reduces energy consumption by **9.9**% on average, up to **28.2**%.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computer systems organization → Architectures**; **Embedded systems**.

ACM Reference Format:
Tommy McMichen, David Dlott, Panitan Wongse-ammat, Nathan Greiner, Hussain Khajanchi, Russ Joseph, and Simone Campanoni. 2025. Saving Energy with Per-Variable Bitwidth Speculation . In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716271

*Both authors contributed equally to this research.

ASPLOS '25, Rotterdam, Netherlands.

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716271

1 Introduction

The fast approaching ubiquity of smart devices depends on the evolution of low energy, general purpose CPUs capable of delivering reasonable performance. Next-generation applications like wearable devices, brain activity trackers, and insect-inspired microbots require ultra-low power processors to provide essential functionality while still maintaining performance to ensure user satisfaction. While specialization in the form of a burgeoning list of fixed [9, 10, 13, 26] and programmable [22, 32, 43] accelerators can provide unparalleled power-performance benefits for dedicated tasks, most systems still rely on general purpose CPUs to perform critical system tasks, such as data transfer between accelerators, file system management and communication with larger networks. These core services are not provided by the various accelerators, but are essential for their function. Providing performant, general-purpose compute capability with tight energy constraints is becoming increasingly difficult as Moore's Law comes to an end.

At this point, most of the obvious inefficiencies at each layer of the system stack have been thoroughly studied. The best hope for unlocking additional energy savings lies in mindful optimizations that span several layers of the system stack. We identify one such opportunity for cross-cutting optimization: *the bitwidth of program variables.*

The bitwidth of program variables has a cascading effect on both the *optimization space available to the compiler* and the *efficient hardware utilization.* In modern embedded programming languages—such as C, C++ and Rust—the bitwidth required for storing each program variable is specified by the programmer. Compilers take this selection for granted and, aside from a few trivial cases, propagate this bitwidth decision during register allocation, where program variables are mapped into registers capable of holding *at least the same number of bits.* As a consequence of this, space in the already-limited register file is reserved for bits that have *no impact on computation.* If there are more variables than available registers at a given program point, the compiler must *spill*, allocating space on the stack and inserting load and store instructions to access spilled variables. These instructions induce communication between the core and its cache, increasing energy consumption from data movement and stall cycles. Additionally, they increase the number of dynamic instructions that must be executed and the number

of static instructions that must be resident in the instruction cache. With this rigid treatment of variable bitwidths, the compiler introduces cascading performance penalties and energy consumption throughout the system.

These performance penalties arise from code developed for the worst-case, using the data type that accommodates the highest possible value for any input. Even worse, programmers with limited time or ability to perform manual code optimizations and tuning are conservative, often using the highest possible bitwidth. This raises the question: *Is the full bitwidth specified for variables by programmers typically needed?* Our results suggest the answer is a resounding ***no***.

This paper introduces BitSpec, a compiler-architecture co-design that speculates on the *expected* bitwidth of integer variables. BitSpec unlocks the potential of the next generation of tiny devices by ***decoupling*** *the programmer-specified bitwidth* of variables from the *actual bitwidth required during execution.* With the BitSpec compiler, no changes to the source code are needed, making BitSpec an *automatic, transparent technique.* The compiler speculates on the expected bitwidth needs of each variable in the program, reducing the bitwidth of variables accordingly. These reduced bitwidth variables are speculatively placed into 8-bit slices of a word-length register, efficiently storing them. Because the run-time values of these variables may exceed the size of their slice, our hardware monitors them and reports misspeculation. Upon misspeculation, the running binary will re-execute the affected region at its original bitwidth, ensuring correctness. This paper makes the following contributions:

- Evaluates the gap between programmer-selected bitwidth and required bitwidth.
- Investigates speculative bitwidth selection techniques.
- Introduces a novel compiler-architecture co-design to efficiently speculate on operations at reduced bitwidths and proves that it maintains program correctness.
- Implements BitSpec on two ARM processors, one conventional and one with state-of-the-art μarchitectural techniques for energy savings on tiny devices.
- Evaluates BitSpec's impact on energy consumption for both processors using gate-level architecture simulation.

2 Bitwidth Selection

The programmer-selected bitwidth for variables has myriad effects on the performance and energy usage of a program. Today, these variables are placed in the register file using *at least* the programmer-specified bitwidth. This leads to inefficient use of the register file, as the programmer-selected bitwidth is *commonly much greater* than bitwidth *required* for computation. Figure 1b categorizes dynamic assignments to integer variables in the LLVM IR by their programmer-selected bitwidth, across benchmarks in the `mibench` suite. We see 56 – 90% of dynamic assignments are made at 32- and 64-bits. However, Figure 1a—which categorizes by the *required bitwidth*—shows that 40 – 100% of instructions need only 8-bits. This stark contrast illustrates the gap between the programmer-selected bitwidth, and the actual bitwidth required for the computation. To describe this gap, we pose *the bitwidth selection problem.*

2.1 The Bitwidth Selection Problem

Informally, the bitwidth selection problem minimizes the number of extraneous bits used to store dynamically computed values, while guaranteeing that the bitwidth is sufficient to store the largest value seen at run-time. To do so, the solution can perform bitwidth selection and other program transformations, so long as correctness[1] is maintained.

Definition. Given a program p, produce an input-output equivalent program p' containing integer variables V with bitwidth selections $BW : V \mapsto \mathbb{N}$, where $\langle v \rangle$ is the sequence of dynamically computed values stored in $v \in V$:

$$\begin{aligned} \underset{BW}{\text{minimize}} \quad & \sum_{v \in V} \sum_{a \in \langle v \rangle} \left(BW(v) - \textbf{RequiredBits}(a) \right) \\ \text{subject to} \quad & BW(v) \geq \max_{a \in \langle v \rangle} \textbf{RequiredBits}(a) \\ \text{where} \quad & \textbf{RequiredBits}(a) \doteq \lfloor \lg(a) + 1 \rfloor \end{aligned}$$

Figure 1a shows the aggregate[2] of **RequiredBits**, while Figure 1b shows the aggregate of BW in the original program.

2.2 Bitwidth Selection with Static Analysis

Multiple approaches to bitwidth selection using static analysis exist [8, 38, 40], the most prominent being demanded bits analysis—an implementation of which is available in LLVM. Figure 1c evaluates LLVM's demanded bits analysis where $BW(v) = \textbf{DemandedBits}(v)$. While the approach does improve upon the programmer-selected bitwidth—increasing the mean usage of 8-bit assignments from 23% to 41%—it fails to achieve the full potential, seen in Figure 1a. In some cases, demanded bits analysis misses *all opportunities*, such as `sha`, where it simply outputs the original bitwidth selection, missing the opportunity for 42% 8-bit utilization.

2.3 Speculative Bitwidth Selection

With the static approach faltering in non-trivial cases, we look down other avenues. Past work has explored speculation at the coarse, basic-block granularity [34]. This solution can be modeled in the bitwidth selection problem where $BW(v) = \max_{w \in \textbf{BasicBlock}(v)} BW(w)$. We evaluate this approach by coercing the bitwidth selection of variables to the maximum bitwidth seen within its basic block, the results of which are shown in Figure 1d. We find that, while in some cases, such as `stringsearch` and `dijkstra`, this coarse-granularity achieves the same selection as the instruction granularity, it misses out on opportunities in many domains. This is primarily seen in security applications and

[1] It is assumed that the original bitwidth selection in p is correct.
[2] All values obtained using the `large` input provided with MiBench.

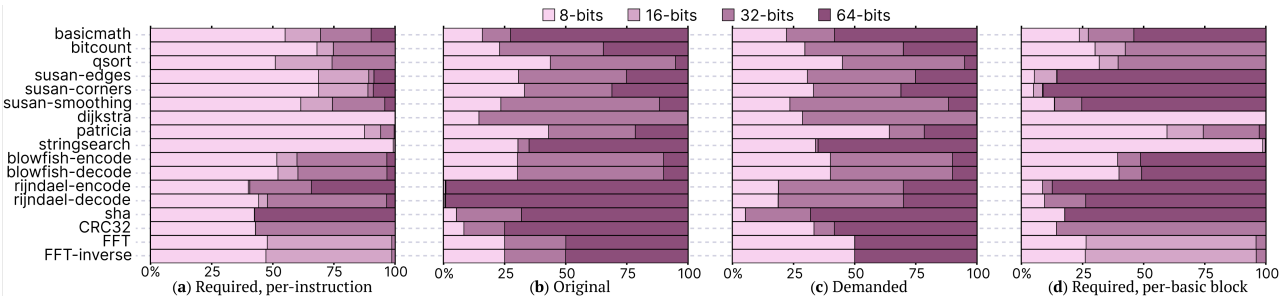

Figure 1. Percentage of dynamic LLVM IR integer instructions at each bitwidth, using different bitwidth selection techniques. No existing approach (**b**-**d**) is capable of reducing bitwidth utilization to the bitwidth actually required for *each* instruction (**a**).

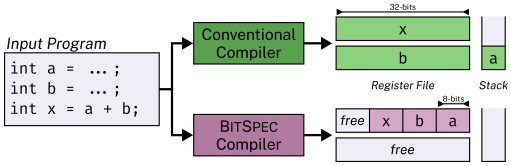

Figure 2. By operating on reduced bitwidth values, BitSpec better utilizes the register file, reducing stack usage.

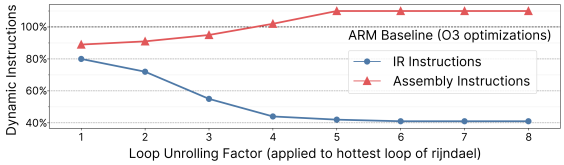

Figure 3. Loop unrolling reduces the number of IR instructions executed, but increases assembly instructions.

all variants of `susan`, where 8-bit utilization is significantly lower. This is especially prevalent on `susan-corners`, whose 8-bit utilization reduces from ~70% to 5%. This results from *few high-bitwidth* variables coercing a *large number of low-bitwidth* variables to their *worst-case bitwidth*.

2.4 Our Approach

Seeing the limitations of prior work, we propose BitSpec: an approach to the bitwidth selection problem via profile-guided, speculative bitwidth selection, per-instruction speculation, and a compiler-architecture co-design to efficiently handle misspeculation. At the core of this approach is the ability to speculatively reduced the bitwidth of *individual variables* for storage *and* computation. Doing so drastically reduces variable spillage, reversing the current that caused the cascade of issues described in §1. In the remainder of this section, we illustrate how BitSpec remedies the ailments of poor bitwidth selection and unlocks new energy savings.

2.5 How Does Bitwidth Selection Impact Efficiency?

As stated in §1, the bitwidth of program variables has a cascading effect on both *efficient hardware utilization* and the *optimization space available to compilers*. In this section we will provide more detailed examples where solving the bitwidth selection problem improves program efficiency.

Better use of the register file. Reducing the bitwidth of program variables reduces cache accesses by fitting more, lower bitwidth, variables into the same register file. Consider the example in Figure 2, where the architecture has two 32-bit registers that can only be accessed at 32-bits, as is the case for ARM-based architectures. In this case only two variables can be mapped to registers at any given time while every other variable is spilled onto the stack. These spills generate additional load and store instructions to access the variables, increasing cache accesses and requiring more instructions be resident in the instruction cache. When the bitwidth of variables is reduced to 8 bits, a new opportunity arises: the compiler can map multiple values into each of these conventional registers. If we are able to reduce variables to 8 bits, we can pack up to four times as many variables into a conventional 32-bit register. This significantly reduces the number of spilled variables, removing costly memory instructions.

Enable conventional compiler optimizations. Compilers rely upon transformations such as loop unrolling and function inlining to unlock additional opportunities for optimization. We refer to these transformations as *expanders*, as they expand the size of the program by instantiating dynamic code paths as static control flow. Figure 3 shows an example of the benefits unlocked by loop unrolling, monotonically reducing the number of dynamic IR instructions executed as loops are unrolled more. However, compilers do not typically unroll loops more than a few times as performance of the assembly quickly degrades. This can be seen in Figure 3, as an unrolling factor of 4 or greater results in an increase in executed assembly instructions. One major cause of this is increased register pressure [42], which has no modern solution aside from increased register file size. BitSpec ameliorates this increased register pressure by packing more variables into the register file, allowing it to reap the benefits.

3 BitSpec: Speculative Bitwidth Selection

To perform fine-grain, speculative bitwidth selection, we propose a combination of profile-guided bitwidth selection with compiler-architecture co-designed speculation Figure 4.

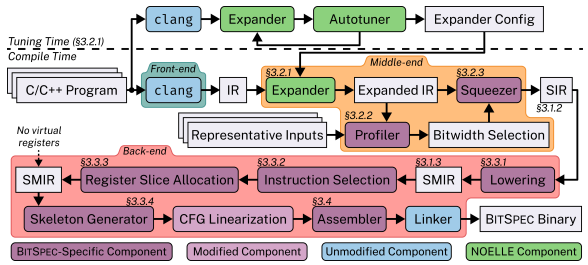

Figure 4. The BitSpec compilation pipeline, specialized components are annotated with their section number.

Before going into detail, we will walk through an example of compiling and running the following program:

```
1 uint32_t x = 0;
2 do { x += 1; } while (x <= 255);
3 return;
```

BitSpec uses the off-the-shelf `clang` front-end to compile C/C++ programs to LLVM's intermediate representation (IR).

```
1 ENTRY: br BODY
2 BODY:
3   %x0 = phi i32 [0, %ENTRY], [%x1, %BODY]
4   %x1 = add i32 %x0, 1
5   %check = cmp ule i32 %x1, 255
6   br i1 %check, label %BODY, label %EXIT
7 EXIT: ret
```

The profiler (§3.2.2) runs the program on representative inputs to generate statistics about the bitwidth needs of each variable. For example, the average number of bits needed to store the result of `x0` and `x1` is 8 bits. These statistics are used to produce a speculative bitwidth selection, in this example we will use the average number of bits.

The squeezer (§3.2.3) transforms variables to utilize their speculative bitwidth selection, and injects software-based speculation handling. Speculation handling is supported by our IR extensions (§3.1), which allow us to specify the basic block to jump into upon misspeculation (line 2).

```
1 ENTRY: br BODY
2 BODY: handler = HANDLER
3   %x0 = phi i8 [0, %ENTRY], [%x1, %BODY]
4   %x1 = add i8 %x0, 1 !speculative
5   %check = cmp ule i8 %x1, 255
6   br i1 %check, label %BODY, label %EXIT
7 EXIT: ret
8 HANDLER:
9   %x2 = zext %x0 to i32
10  br BODY2
11 BODY2:
12  %x3 = phi i32 [%x2, %HANDLER], [%x4, %BODY2]
13  %x4 = add i32 %x3, 1
14  %check2 = cmp ule i32 %x4, 255
15  br i1 %check2, label %BODY2, label %EXIT2
16 EXIT2: ret
```

Finally, the back-end (§3.3) lowers the IR to a binary, using our ISA extensions (§3.4) and generates code necessary for the μarchitecture (§3.5) to detect misspeculation and redirect execution to the handler. When our example program is run, we get the following execution state at the end of each dynamically executed basic block (empty cells are undefined):

time ↓

Block	x0	x1	x2	x3	x4
ENTRY					
BODY	0	1			
BODY	1	2			
⋮	⋮	⋮			
BODY	254	255			
BODY	255	misspec!			
HANDLER	255		255		
BODY2			255	255	256
EXIT2					

When is BitSpec applicable? Broadly, BitSpec benefits programs where programmers cannot safely reduce the bitwidth of variables due to outliers or uncertainty in the range of input values. An example of this is `stringsearch`, whose hot code is shown in Listing 1. In this code, the majority of operations are on values of type `size_t` (64 bits on our target architecture), reflecting Figure 1b. However, in the provided input, the maximum length of `pat` and `str` are 12 and 56, respectively, which can be stored in 8 bits. By speculating on the bitwidth of these variables, the function can be transformed to speculatively operate on entirely 8-bit integers. There is a similar pattern in `CRC32`, where the length of each line in the provided input file varies from 0 to 2729, with an average of 145.8. In this case, the majority of execution can occur at 8-bits, with speculation handling the outliers where $length > 255$.

Listing 1. Hot code from `stringsearch`.

```
1 char *strsearch(char *pat, char *str) {
2   size_t patlen = strlen(pat); // 0 < patlen <= 12
3   size_t strlen = strlen(str); // 0 < strlen <= 56
4   size_t pos = patlen - 1;     // 12 <= pos <= 56
5   while (pos < patlen) {
6     while (pos < strlen && table[str[pos]] > 0)
7       pos += table[str[pos]];
8     ... } ... }
```

When is BitSpec limited? Some code patterns are not amenable to the BitSpec execution model due to design decisions. Small, recursive functions exacerbate the cost of misspeculation where, in the worst case, the function body will be executed twice per invocation. At the other extreme, large functions lead to missed opportunities because, following a single misspeculation, the remainder of the function is executed at its original bitwidth.

3.1 Representing Speculation in the Compiler

The BitSpec compiler speculatively lowers the bitwidth of variables. As a consequence, the compiler needs to differentiate between variables whose bitwidths are speculative and those that are not. Today, compiler intermediate representations provide no support for representing this type of speculation. To this end, we introduce *speculative regions*, a minor addition to the compiler's intermediate representation. While it can be generalized, we will describe it in the context of LLVM's middle-end IR [18] and back-end Machine IR (MIR) [19]. We refer to our extended forms as Speculative IR (SIR) and Speculative Machine IR (SMIR)[3].

[3]Pronounced *seer* and *smear*, respectively.

3.1.1 Speculative Regions.

A speculative region (SR) is a single entry, single exit [25] sequence of basic blocks (BB). **Entry** : $SR \rightarrow BB$ gives the first basic block in the sequence. The shaded box containing `B.nonphis` at the bottom of Figure 6 is an example of a speculative region. Each speculative region has a single *handler*, i.e., a basic block that will be invoked iff an instruction in the speculative region misspeculates, which can be queried: **Handler** : $SR \rightarrow BB$. A basic block can only be the handler for a single speculative region. A handler cannot be contained within a speculative region. Finally, handlers cannot be the target of any branches, as they can only be entered upon misspeculation.

Conceptually, speculative regions can be thought of as try blocks in exception handling (EH). However, in practice, exceptions can only be thrown between functions, which is too coarse grained for our use case. One could implement speculative regions using existing EH support in LLVM, by outlining them into their own function. However, this introduces extra code that, in our prototype, we could not safely remove without inducing a large overhead.

3.1.2 Speculative IR (SIR)

extends LLVM IR by introducing speculative regions and modifying how predecessors of basic blocks are computed. SIR solely extends LLVM IR, so any valid LLVM IR program is a valid SIR program. To allow for usage of previously-defined variables in handlers—and prohibit usage of possibly misspeculated variables—their predecessors are computed as follows:

$$\textbf{Preds}(\textbf{Handler}(SR)) = \textbf{Preds}(\textbf{Entry}(SR)) \tag{1}$$

Figure 6 demonstrates: `handler.B` has `B.phis` as a predecessor without being the target of a branch from it. This predecessor relation will be used to prove that all values from the current speculative region will be dead upon entering the handler.

3.1.3 Speculative Machine IR (SMIR)

extends SIR:

1. SMIR contains both virtual and physical registers.
2. SMIR contains machine-specific operations.

Since physical registers in SMIR are not SSA, they need to be properly managed by the register allocator in case of misspeculation. To accomplish this, the predecessors of a misspeculation handler in SMIR are computed as:

$$\textbf{Preds}(\textbf{Handler}(SR)) = \bigcup_{MBB \in SR} MBB \tag{2}$$

3.2 Speculatively Reducing the Bitwidth of Variables

The BitSpec compiler's middle-end operates on SIR and consists of the *expander*, *bitwidth profiler* and *squeezer*.

3.2.1 Expander.

Per §2.5, expanding the code unlocks compiler optimizations. To exploit this, we introduce the *expander*, a middle-end compiler pass that aggressively applies function inlining and loop unrolling using NOELLE [30].

To explore the limit of expansion the baseline architecture is capable of handling, we tune the expander with an auto tuner [2] to achieve the best performance—reduction in dynamic instructions executed—on the baseline architecture. The search space of the auto tuner is: *unrolling factor*, *max function size*, and *max loop size*. The unrolling factor dictates the max number of times any loop in the target program will be unrolled. The max function size and max loop size dictate the max number of static instructions allowed in any function or loop when unrolling and inlining. We perform this tuning offline—a total of 10 days for our evaluation (§4)—with a single output configuration for all benchmarks.

3.2.2 Bitwidth Profiler.

The profiler begins by gathering bitwidth utilization statistics. For each variable, we record the maximum (MAX), minimum (MIN) and average (AVG) bitwidth required during execution. The average bitwidth requirement of a SIR variable is computed as follows:

$$\text{AVG}(v) = \frac{1}{|\langle v \rangle|} \sum_{a \in \langle v \rangle} \textbf{RequiredBits}(a)$$

With these statistics, the profiler produces *target bitwidth selections* $T : V \rightarrow \mathbb{N}$. In this paper we explore $T = MAX$, $T = AVG$, and $T = MIN$ as heuristics. By varying the heuristic, we tune *aggressiveness*: a heuristic is more aggressive if it produces lower bitwidth selections. Figure 5 shows the aggregate bitwidth selections of our heuristics.

However, T cannot be directly used as bitwidth selections for the program. There are a few reasons for this. First, similarly to LLVM IR, SIR requires that all operands of an instruction have the same bitwidth. Second, not all instructions can be directly mapped to speculative operations in the ISA, the existence of an operation can be queried with **Speculative**?. Third, not all instructions are idempotent: they cannot be executed any number of times without side effects. Idempotency of an instruction can be queried with **Idempotent**?, discussed in §3.2.3. Finally, we must be able to extend the result to its original bitwidth, $O : V \rightarrow \mathbb{N}$, and obtain the same value that the original program would have computed. To accommodate these caveats, we will apply constraints to T to produce the bitwidth selection $BW : V \rightarrow \mathbb{N}$.

We apply these constraints with the **Squeezable**? relation, where **Def** maps variables to their defining instruction.

$$\begin{aligned} \textbf{Squeezable}?(v = op\ v_0, \ldots v_n) \iff\ & \textbf{Speculative}?(op) \\ & \wedge\ \textbf{Idempotent}?(\textbf{Def}(v)) \\ & \wedge\ \textbf{extend}\ v\ \textbf{to}\ O(v) \equiv \textbf{Orig}(v) \end{aligned} \tag{3}$$

We then compute the bitwidth selection, BW:

$$BW(v) = \begin{cases} \max\left(T(v), \max_{u \in \textbf{Operands}(i)} T(u)\right) & \textbf{Squeezable}?(i) \\ O(v) & \text{otherwise} \end{cases}$$

$$\text{where } i = \textbf{Def}(v)$$

3.2.3 Squeezer.

With the profiler's bitwidth selection BW in hand, the compiler then invokes the squeezer, which speculatively reassigns the bitwidth of variables and injects the appropriate misspeculation handling. The squeezer creates

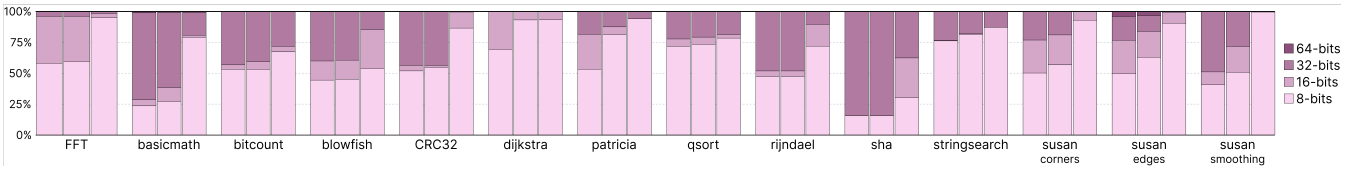

Figure 5. Percent of dynamic integer instructions the profiler classifies as 8, 16 or 32 bits when $T = MAX, AVG, MIN$, respectively.

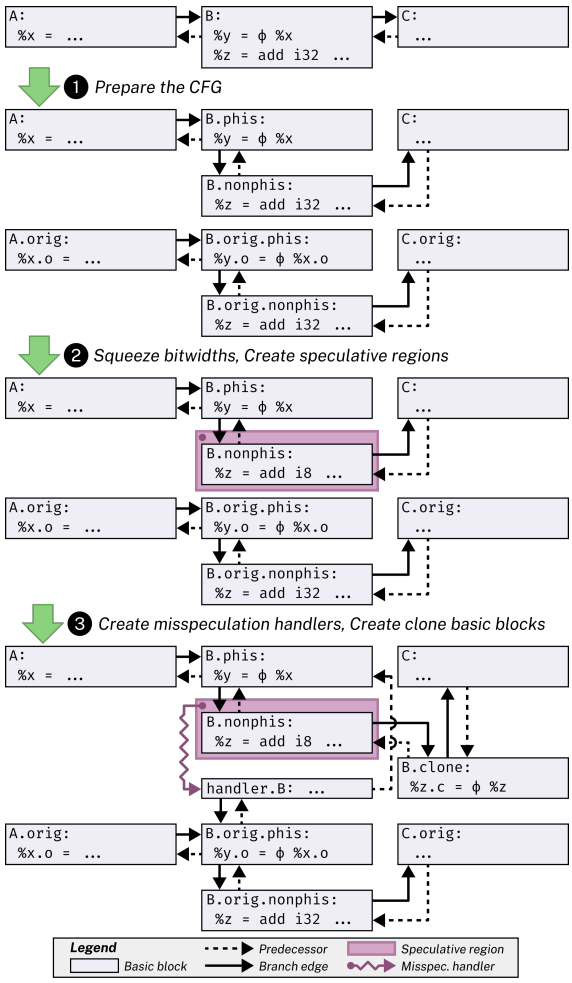

Figure 6. The squeezer takes a function (top) and applies ❶ – ❸. Note that after ❸ `B.phis` is the sole predecessor of `handler.B`, from the extended definition of predecessors in SIR(§3.1.2).

a function where, once misspeculation occurs, live variables are extended to their original bitwidth and the basic block where misspeculation occurred is re-executed at its original bitwidth. Following this, execution resumes at the original bitwidth specified by the developer until the end of the function. A visualization of each pass taken by the squeezer can be seen in Figure 6. The squeezer operates in three passes:

❶ Prepares the control-flow graph (CFG).
❷ Speculatively reduces the bitwidth of variables, inserting speculative regions as needed.
❸ Inserts handlers for each speculative region.

❶ **CFG Preparation.** To prepare the program, we transform the CFG [1] of each function.

First, basic blocks are split such that, $\forall BB \in Function$:

$$|\{l \in BB \mid l \textbf{ isa load}\}| \geq |\{s \in BB \mid s \textbf{ isa store}\}| = 0 \\ \vee |\{s \in BB \mid s \textbf{ isa store}\}| \geq |\{l \in BB \mid l \textbf{ isa load}\}| = 0 \tag{4}$$

$$|\{i \in BB \mid i \textbf{ is volatile}, i \textbf{ isa call}\}| = |BB| = 1 \\ \vee |\{i \in BB \mid i \textbf{ is volatile}, i \textbf{ isa call}\}| = 0 \tag{5}$$

$$|\{p \in BB \mid p \textbf{ isa } \phi\}| = |BB| \\ \vee |\{n \in BB \mid \neg(n \textbf{ isa } \phi)\}| = |BB| \tag{6}$$

To safely re-execute basic blocks upon misspeculation, we must guarantee that speculation only occurs in *idempotent* basic blocks. We do this as prescribed by prior work [12], splitting basic blocks such that no write-after-read (WAR) dependencies exist within a basic block with equation (4). Beyond WAR dependencies, a basic block is non-idempotent iff it contains a non-idempotent instruction, e.g. I/O operations, which are represented in SIR as either volatile instructions or calls. Equation (5) guarantees that basic blocks contain either idempotent or non-idempotent instructions. Therefore, we can query whether or not a basic block is idempotent:

Idempotent?$(BB) \iff \forall i \in BB, \neg(i \textbf{ is volatile} \vee i \textbf{ isa call})$

Equation (6) ensures that basic blocks contain either ϕ or non-ϕ instructions, but not both. This guarantees that no ϕ instructions are involved in misspeculation handling except for those injected by ❸, simplifying the transformation.

Finally, we transform the CFG to contain two disjoint sub-graphs: CFG_{orig}, containing all variables at their original bitwidth, and CFG_{spec}, containing variables assigned to their speculative bitwidth. First, we clone all basic blocks in the CFG, creating the set of originals CFG_{orig} and the set of clones CFG_{spec}. **Spec** : $CFG_{orig} \mapsto CFG_{spec}$ maps each original basic block, instruction and variable to its clone. The inverse, **Orig** : $CFG_{spec} \mapsto CFG_{orig}$ is also created. We then replace the uses of all variables and basic blocks in CFG_{spec} with their clone using **Spec**. This guarantees that no variable defined in CFG_{spec} is used in CFG_{orig}, and vice versa, with no control path existing between them.

❷ **Speculatively reduce the bitwidth of variables.** The squeezer then uses the bitwidth selections, BW, produced by the profiler to transform CFG_{spec}. For each variable v, we speculatively cast the operands of its defining instruction to $BW(v)$, producing $op'_1, ..., op'_n$. Then, a new definition v' is created with bitwidth $BW(v)$, taking $op'_1, ..., op'_n$ as operands. All uses of v are replaced with v'. If one does not already

exist, a speculative region SR is created for the basic block containing v' and the basic block is inserted into SR. After applying these steps to all speculative instructions, the original instructions and extraneous speculative truncates are removed via a simple dead code elimination.

❸ **Misspeculation Handling** is then inserted. First, a new basic block $BB_{handler}$ is inserted for each speculative region created by ❷ and registered as its handler. The terminator of $BB_{handler}$ is set to an unconditional branch to $BB_{orig} = \textbf{Orig}(BB_{spec})$, guaranteeing:

$$\forall s \in \textbf{Succ}(BB_{handler}), s \in CFG_{orig} \tag{7}$$

Let V_{orig} be the set of variables live at the beginning of BB_{orig}. For each variable in V_{orig}, create an extension of its speculative clone to its original bitwidth in $BB_{handler}$, storing a mapping from the extension to the clone in $\textbf{Extended} : V \mapsto V$. Then, a ϕ instruction is created in BB_{orig}, such that:

$$\forall v_{orig} \in V_{orig}, v_{ext} = \textbf{extend}\ v\ \textbf{to}\ O(v),$$

$$v_\phi = \phi \begin{pmatrix} v_{ext} & BB_{handler} \\ v_{orig} & BB_{pred} \end{pmatrix} \tag{8}$$

$$\text{where } v = \textbf{Spec}(v_{orig})$$

$$\text{and } \textbf{Pred}(BB_{orig}) = \{BB_{pred}\}$$

All uses of v_{orig} dominated by BB_{orig} are replaced with v_ϕ.

Additionally, create a basic block BB_{clone} to hold clones of all variables defined within the speculative region. The terminator of BB_{clone} is set to that of BB_{spec} and the terminator of BB_{spec} is set to an unconditional branch to BB_{clone}. For each variable $v \in BB_{spec}$ live at the end of BB_{spec}, create v_{clone}, a ϕ instruction in BB_{clone} that takes v as its value for the edge from BB_{spec}. Replace all uses of v outside BB_{spec} with v_{clone}. This guarantees that all variables defined in BB_{spec} are only used within BB_{spec} or on the exiting branch edge to BB_{clone}:

$$\forall u \in \textbf{Uses}(v), u \in BB_{spec} \vee u \in \textbf{Succ}(BB_{spec}),$$
$$\text{where } \textbf{Succ}(BB_{spec}) = \{BB_{clone}\} \tag{9}$$

We will now prove that the above transformations preserve program correctness. Informally, we will demonstrate that, by construction, no variable defined within a speculative region can be used by its handler nor CFG_{orig}.

Theorem 3.1. *Any variable defined within a speculative region is dead at the beginning of its handler.*

Proof. Given a variable v_{spec} within basic block BB_{spec} within speculative region SR where $\textbf{Handler}(SR) = BB_{handler}$ and $\textbf{Succ}^+$ is the transitive closure of **Succ**. By equation (7):

$$BB_{spec} \notin CFG_{orig},\ \nexists s \in \textbf{Succ}(BB_{handler}) \mid s \notin CFG_{orig},$$
$$\implies BB_{spec} \notin \textbf{Succ}^+(BB_{handler})$$

By equation (9):

$$BB_{handler} \neq BB_{spec},\ BB_{handler} \notin \textbf{Succ}(BB_{spec})$$
$$\implies \forall u \in \textbf{Uses}(v_{spec}), u \notin BB_{handler}$$
$$\implies \textbf{Uses}(v_{spec}) \cap BB_{handler} = \emptyset$$

By the definition of liveness:

$$\textbf{LIVE}_{\textbf{in}}[n] = \{v \mid u \in \textbf{Uses}(v) \wedge u \in n\}$$
$$\cup \left(\left(\bigcup_{s \in \textbf{Succ}(n)} \textbf{LIVE}_{\textbf{in}}[s] \right) \setminus \{v \mid \textbf{Def}(v) \in n\} \right)$$
$$\implies \nexists v \in \textbf{LIVE}_{\textbf{in}}[\textbf{n}] \mid v \notin \textbf{Succ}^+(n) \wedge \textbf{Uses}(v) \cap n = \emptyset$$
$$\implies v_{spec} \notin \textbf{LIVE}_{\textbf{in}}[BB_{handler}]$$

For a variable v to be dead at the beginning of BB:

$$v \notin \textbf{LIVE}_{\textbf{in}}[BB]$$

Therefore, v_{spec} is dead at the beginning of $BB_{handler}$. □

We next prove that the state of variables at the end of re-execution matches that of the original execution. This is done by showing that all ϕ instructions inserted in ❸ will take on an equivalent value for all incoming edges.

Theorem 3.2. *The state of variables after executing the ϕs of a basic block in CFG_{orig} is equivalent for all predecessors.*

Proof. Given speculative region SR with basic block BB_{spec} where $\textbf{Handler}(SR) = BB_{handler}$ and $\textbf{Succ}(BB_{handler}) = \{BB_{orig}\}$. By equations (3), (6) and (8) and theorem 3.1:

$$\forall v_{ext} \in BB_{handler},\ v_{ext} \equiv v_{orig},$$
$$\forall v_\phi \in \textbf{Phis}(BB_{orig}),\ v_\phi = \begin{cases} v_{ext} & BB_{handler} \\ v_{orig} & BB_{pred} \end{cases}$$
$$\text{where } v = \textbf{Extended}(v_{ext}) = \textbf{Spec}(v_{orig})$$
$$\text{and } \textbf{Pred}(BB_{orig}) = \{BB_{handler}, BB_{pred}\}$$
$$\implies \forall v_\phi \in \textbf{Phis}(BB_{orig}), v_\phi(BB_{handler}) \equiv v_\phi(BB_{pred})$$

For the variable state of a basic block following ϕ execution to be equivalent along all predecessors:

$$\forall v_\phi \in \textbf{Phis}(BB),\ v_\phi(BB_0) \equiv ... \equiv v_\phi(BB_n)$$
$$\text{where } \textbf{Preds}(BB) = \{BB_0, ..., BB_n\}$$

Therefore, the variable state of BB_{orig} following ϕ execution is equivalent for all incoming control edges. □

3.2.4 Optimizing with Speculation.

With explicit speculation in SIR, we can enable new optimizations. To explore this, we use speculation *eliminate compare instructions*.

Given a compare instruction, that takes a speculative variable v and a constant integer c as operands, if c cannot be stored in $BW(v)$, i.e. $c \geq 2^{BW(v)}$, the compare can be optimized to rely on the result of speculation on v. If v does not misspeculate, then $v < 2^{BW(v)}$, and the compare can be replaced by a constant `true` or `false` based on the operator. Because the compare now relies on this speculation result, v cannot be removed from the function.

3.3 Back-end

The back-end of the BitSpec compiler is responsible for generating an assembly program from a SIR program. Register allocation accounts for the possibility of misspeculation within speculative regions. To leverage the capabilities of a compiler-architecture co-design, the back-end also optimizes code layout to reduce misspeculation overhead.

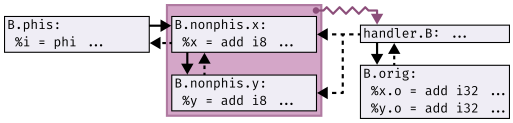

Figure 7. SMIR exposes possible control flows caused by misspeculation to the register allocator.

3.3.1 Lowering SIR to SMIR is similar to LLVM's IR to MIR lowering, but is extended to propagate speculative regions and perform some normalization. Each SIR basic block (*BB*) maps to many SMIR machine basic blocks (*MBB*). Each *MBB* maps to one *BB*, queried with **BasicBlock** : $MBB \longrightarrow BB$. Speculative regions are propagated such that:

$$MBB \in SR \iff \textbf{BasicBlock}(MBB) \in SR$$

Each speculative region is then normalized by splitting its *MBB*s such that each contains only a single instruction.

3.3.2 Instruction Selection ensures that speculative instructions will be properly monitored by the hardware by mapping them to speculative operations. For each variable v defined in a speculative region, where $BW(v) < O(v)$, we map its definition to its speculative operator—which must exist due to the **Squeezable**? relation.

3.3.3 Register Allocation maps all virtual registers to physical registers or slices of physical registers. We use existing LLVM infrastructure by exposing all slices to LLVM as subregisters of each 32-bit register. The predecessor relation of SMIR ensures that this mapping is correct in the case of misspeculation. Figure 7 shows this, as each MBB within an SR contains only one instruction, exposing any potential control flow caused by misspeculation to the register allocator. Finally, we apply a constraint to spilling: each spill instruction within an SR must be placed at the top of its MBB. Without this constraint, misspeculation could result in spills not executing before jumping to the handler.

3.3.4 Code Layout for Efficient Speculation. Next the CFG is linearized to a sequence of instructions. A naïve linearization incurs heavy performance penalties so we optimize code layout—leveraging our compiler-architecture co-design—to achieve efficient misspeculation handling with minimal overhead for speculative code. In this design, the hardware increments the program counter (PC) by a constant Δ when it detects misspeculation. The compiler selects a value for Δ and creates a code layout guaranteeing that PC + Δ will cause execution to enter the current speculative region's handler, maintaining the SMIR semantics.

To achieve this, the compiler generates *skeleton blocks*. For each $MBB \in SR$, the *skeleton generator* creates a single machine basic block MBB_{skel}, containing an unconditional branch to **Handler**(SR). A constraint is added to CFG linearization, such that, $MBB_{skel} = MBB + \Delta$. We layout all MBB_{spec}, where **BasicBlock**(MBB_{spec}) $\in CFG_{spec}$, contiguously and set Δ to the size of that contiguous region.

Instruction	Pseudocode	Misspec?
Addition	$B_d := B_n + (B_m \text{ or } imm4)$	Overflow
Subtraction	$B_d := B_n - (B_m \text{ or } imm4)$	Underflow
Logic	$B_d := B_n \text{ op } (B_m \text{ or } imm4)$	Never
Comparison	$\text{cond}(B_n \text{ op } (B_m \text{ or } imm4))$	Never
Spec. Load	$B_d := Mem_{32}[R_n + (B_m \text{ or } imm8)]$	BW(load) > 8
Load	$B_d := Mem_8[R_n + (B_m \text{ or } imm8)]$	Never
Store	$Mem_8[R_n + (B_m \text{ or } imm8)] := B_d$	Never
Extension	$R_d :=$ Sign/ZeroExtend(B_n)	Never
Spec. Trunc.	$B_d :=$ Truncate(R_n)	BW(R_n) > 8

$imm4/8$ is a 4/8-bit immediate R/B_i is a 32/8-bit register (slice)
Mem_n is a n-bit memory access

Table 1. We implement instructions as detailed in the pseudocode, which misspeculate under the **Misspec?** condition.

3.4 ISA Support

We extend the ARMv7 ISA to provide speculative, 8-bit variants of existing operations, a summary of which is shown in Table 1. To accommodate our new instructions, we remap ALU operations with a rotated second operand and coprocessor load/stores, effectively removing those instructions.

To support pre-compiled code, we add the ability to enable/disable our instruction remapping. When disabled, the processor runs in *classic mode*, preserving all ARMv7 instructions. Before calling a pre-compiled function, the compiler injects a special assembly instruction which switches to classic mode. Upon returning from the function, the compiler injects an assembly instruction to switch to BitSpec mode.

3.5 Microarchitecture

We modify the μarchitecture to enable register slice access, reduced bitwidth ALU operations, and misspeculation detection. Register slices are supported with modifications to the register file. In the baseline ARM system only 32-bit access to general-purpose registers (GPRs) is allowed. We add support for accessing 8-bit slices of any GPR. This is done by adding byte-level read/write enable signals and modifying forwarding logic to detect dependencies between register slices. Our chosen platform—a low-power, in-order embedded CPU—has no register renaming, making this modification simple. The ALU is segmented into 8-bit slices and operands are routed to their corresponding ALU slice. ALU logic is extended to detect misspeculation using carry/overflow signals at slice boundaries. In the case of misspeculation, the result is not written to the register file and the PC is incremented by a 32-bit special register, which holds Δ (§3.3.4).

4 Evaluation

We implemented the BitSpec compiler in the production-quality LLVM compiler [27]. We utilize a gate-level implementation of our processor to provide accurate energy results. We evaluate on workloads from the MiBench [23] suite.

Our evaluation answers the following research questions:

RQ0 Does BitSpec reduce energy consumption?
RQ1 Does BitSpec improve register allocation?
RQ2 Is speculation necessary?
RQ3 What is the impact of BitSpec-specific optimizations?
RQ4 Does BitSpec enable more aggressive, conventional compiler optimizations?
RQ5 What is the impact of aggressive bitwidth selections?
RQ6 Is BitSpec resilient to alternate inputs?
RQ7 Does BitSpec eliminate the need for programmer-selected bitwidth entirely?
RQ8 Does BitSpec compose with state-of-the-art architecture research for energy savings on tiny devices?
RQ9 How does BitSpec compare to reduced-bitwidth ISAs in production?

4.1 Experimental Setting

We implement our compiler on LLVM 10.0.0—with NOELLE abstractions [30]—by adding middle-end passes, extending the ARM back-end, with no changes to the front-end. We implement the expander using off-the-shelf tools from the NOELLE compiler infrastructure. Speculative regions and their handlers (§3.1.1) are implemented as metadata in LLVM IR and MIR programs. Our BitSpec compiler does not support separate compilation, but it does support precompiled libraries as described in §3.4. We use gllvm to obtain a single bitcode file for the program, not including precompiled libraries, and use this as the input program for our compiler.

We model a 32-bit 6-stage, single-issue, in-order ARMv7 pipeline with 8KiB 4-way instruction and data caches, backed by a 256KiB L2 cache. We implement this pipeline in RTL Verilog and synthesize it to a 45nm gate-level implementation operating at 1.2V using the Synopsys Design Compiler. This processor has been utilized in prior works to demonstrate energy reduction techniques in tiny devices [15, 16, 24], which we build upon. We implement two variants of this processor. Baseline: the processor as described. BitSpec: the processor with the BitSpec ISA and μarchitecture extensions. We present metrics relative to Baseline, unless stated otherwise.

We develop a sample-based simulation flow [44] coupling our gate-level implementation—to accurately track the power consumption and circuit-level timing—with fast functional simulation via Gem5 [7]. We simulate the memory hierarchy with a custom cache model integrated with DRAMSim [29] to build a detailed, complete system performance model. We use SimPoints [33] with a 1M instruction interval size.

We evaluate on a subset of the MiBench suite [23], excluding benchmarks that we could not compile to a whole-program representation with LLVM. The MiBench suite showcases fundamental tasks in a variety of embedded domains, closely matching our target domain discussed in §1.

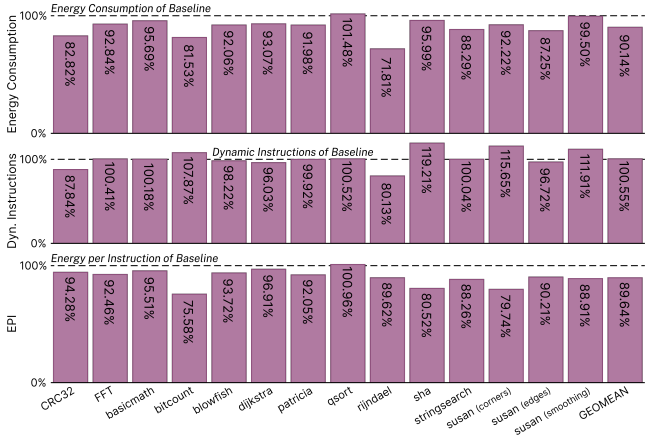

Figure 8. BitSpec reduces *energy consumption* and *energy per instruction (EPI)*, with varying impact on *dynamic instructions*.

RQ0: Reducing Energy Consumption

We use our gate-level power model and architecture simulator to obtain energy results for each benchmark, shown in Figure 8. BitSpec reduces energy consumption by 9.9% on average, with a maximum reduction of 28.2% on `rijndael`.

To understand the source of energy savings, we measured the number of dynamic instructions for each benchmark (Figure 8) and compute the energy per instruction (EPI) from it (Figure 8). Dynamic instruction reduction illuminates when eliminating instructions from the program drives our energy savings. BitSpec has varying impact on dynamic instructions in our benchmarks. We also look at EPI to investigate efficiency, where a lower EPI—without increasing dynamic instructions—implies that we are executing more efficient instructions. BitSpec reduces the EPI across all benchmarks, except `qsort`, showing that we are also producing executables that more efficiently use the available hardware.

To provide a more granular view, we utilize activity counters from the architecture simulator to breakdown the energy consumed by each component. Figure 9 shows the results for the ALU, register file, data cache (D$), and instruction cache (I$). Energy consumed by the rest of the processor is reported as *pipeline*, primarily caused by stall cycles.

Investigating these results, we see a correlation between dynamic instruction reduction and I$ energy reduction, especially for `CRC32` and `rijndael`, as having to fetch fewer instructions from the cache will directly reduce the amount of energy used to access the cache. We will contextualize further investigation into the factors behind the energy savings of BitSpec in terms of the aforementioned metrics.

RQ1: Improving Register Allocation

Following up on our discussion in §1 and §2.5, we investigate the impact of BitSpec on spill instructions. Figure 10 shows the dynamic loads, stores and copies injected by the register allocator, normalized to their sum on Baseline.

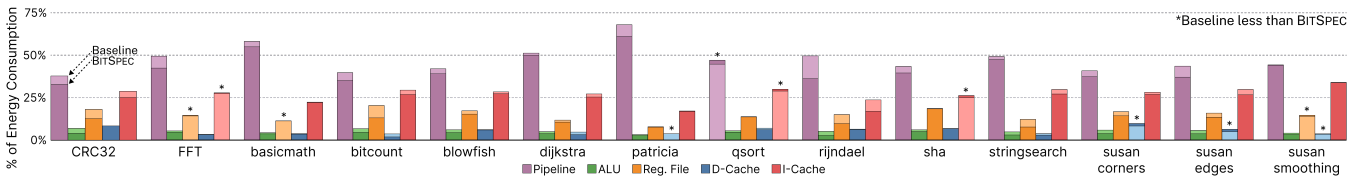

Figure 9. For most benchmarks, BitSpec reduces *energy consumption across all components*, with a few exceptions.

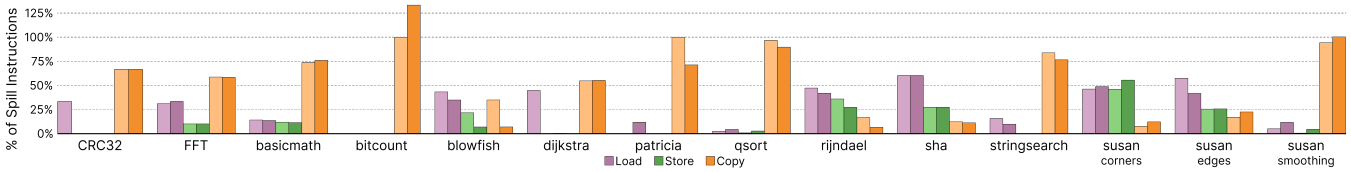

Figure 10. BitSpec reduces spilling-related loads and stores, sometimes trading off this reduction for an increase in register-register copies.

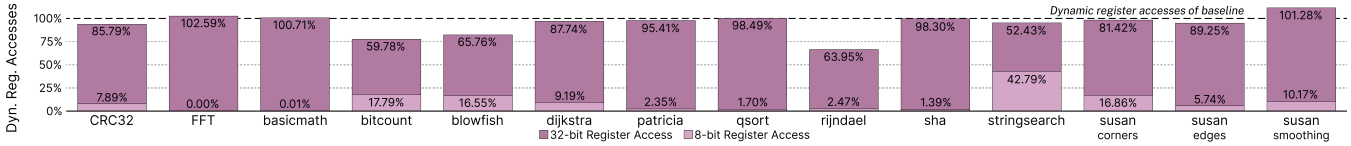

Figure 11. BitSpec reduces the number of dynamic *register accesses*, eliminating some and converting others to 8-bit register slice accesses. All register accesses in Baseline are 32-bits.

For the majority of benchmarks, BitSpec reduces or entirely eliminates spilling-related loads (`CRC32` and `dijkstra`). This is driven by packing multiple, 8-bit variables into a single 32-bit register—allowing more live variables to be resident in the register file—reducing the number of variables that need to be spilled and reloaded. To investigate further, Figure 11 reports the number of dynamic register accesses at 8- and 32-bits. We see that, for the majority of benchmarks, the number of dynamic register accesses decreases and the program makes significant use of 8-bit register slices. Reducing the number of register accesses reduces the energy consumption of the register file. Furthermore, our gate-level model reports that 8-bit register slice accesses incur 1/4 the energy of a 32-bit register access. Because load instructions must go to, at least, the D$—inducing stalls—removing them reduces both D$ and pipeline energy consumption.

Aside from storing and loading variables from the stack, the register allocator manages the placement of variables within the register file, moving variables from one register to another at times. While advanced register allocation techniques minimize these moves by coalescing [21] when possible, they cannot be entirely eliminated. In many cases BitSpec reduces the number of copies, as register slices grant more degrees of freedom to the allocator. However, in `dijkstra` and `susan-edges`, the register allocator trades additional copies for reduced loads, a preferable tradeoff as moves are typically more efficient than loads, improving EPI. In the case of `bitcount`, we see that the compiler trades additional copies for more efficient, 8-bit instructions.

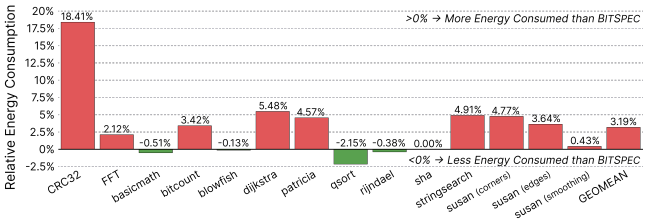

Figure 12. Energy consumption without speculation compared to BitSpec (lower is better). Without speculation, the system consumes an additional 3.19% energy, on average.

RQ2: Register Packing without Speculation

In §2.2 we identified a significant gap between static analysis and actual bitwidth requirements, leading us to investigate speculation. We evaluate the necessity of speculation by comparing with a variant of BitSpec where the compiler performs *no speculation*, with no changes to the ISA or μarch.

Figure 12 shows that, without speculation, the system misses out on an additional 3% energy savings, on average. On `CRC32`, the system achieves *no energy reduction* without speculation. However, we do see the cost of speculation outweigh the benefits of reduced bitwidth computation on `qsort`. In `qsort`, this is caused by the comparison function effectively being executed twice upon misspeculation.

RQ3: Impact of Optimizations

We implement two BitSpec-specific optimizations in our compiler: *compare elimination* and *bitmask elision*. We perform an ablation study on each optimization in turn.

	CRC32	FFT	basicmath	bitcount	blowfish	dijkstra	patricia	qsort	rijndael	sha	stringsearch	susan-edges	susan-corners	susan-smoothing
MAX	0	1	0	0	1	0	0	0	1	0	0	3	3	1
AVG	0	1	1	0	1	200	1	0	1	0	0	5	5	1
MIN	1	1	1	1	81710	200	7238	0	3	50745	1	6	6	2

Table 2. The number of misspeculations increases with more aggressive bitwidth selection heuristics.

Compare elimination (§3.2.4) removes compare instructions based on the result of speculation. Without this optimization, energy consumption of `dijkstra` increases by 9.5% due to a 13.1% increase in dynamic instructions.

Bitmask instructions—which extract a subset of the bits from a variable—can be optimized to operate directly on register slices. For example, `R2 = and R1 0xFF` can be replaced with a move of the lower 8-bit slice of `R1` to `R2`. This is a very common pattern for encoding algorithms, such as `blowfish` and `rijndael`. In addition to removing instructions, this enables further use of variables at 8-bits by other speculative 8-bit instructions. Removing this optimization increases the energy consumption of `blowfish` and `rijndael` by 6.3% and 33.4% relative to Baseline, respectively. For `rijndael` this large impact is a result of optimizing the hottest loop—where all uses of a bitmask instruction can be replaced with a register slice—reducing dynamic instructions by 26.1%.

RQ4: Aggressive Expansion

In §2.5, we motivated the capabilities of BitSpec to ameliorate the drawbacks of aggressively expanding the program and introduced the expander (§3.2.1) to leverage these opportunities. To understand its impact, we run all benchmarks with the expander disabled. Figure 13 shows the results relative to Baseline with the expander enabled. When disabled, Baseline sees a ~10% increase in energy consumption.

While, as discussed in the motivation, we expect to see the expander push the register allocator past its limits and cause a performance degradation, we do not see that in our evaluation. This is due to our autotuning procedure, which targets Baseline instead of BitSpec. However, we do see that the expander trades off optimized code for increased register pressure, which does not result in performance degradation but does diminish its returns. Looking at Figure 13, we see that BitSpec reduces the average EPI by 10.36%, but only 6.41% when the expander is disabled. The magnitude of this difference indicates that BitSpec is able to eliminate loads, improving the efficiency of instructions. This ameliorates the register pressure and allows the expander to realize more of its potential. Thus, we conclude that the benefits of the expander compound with the benefits of BitSpec and that their combination is more than the sum of its parts.

RQ5: Aggressive Bitwidth Selection

§3.2.2 introduced multiple bitwidth selection heuristics, i.e., *MAX*, *AVG* and *MIN*. Figure 14 shows the energy results for each heuristic. For most benchmarks, *MAX* provides the lowest energy consumption, with the exception of `FFT` and `patricia` where the lowest is *AVG* and *MIN*, respectively.

To understand these results, we look at the misspeculation counts in Table 2. More aggressive heuristics tend to increase the misspeculation count, *always* correlating with an increase in energy consumption. This suggests that the cost of misspeculation outweighs further reducing bitwidth selection (see Figure 5). We believe this is an artifact of BitSpec's misspeculation handling, which allows a function to misspeculate only once per invocation before executing the remainder at the original bitwidth. So, if you misspeculate early, all benefits of BitSpec are lost. An alternative that returns to speculative execution would allow aggressive heuristics to balance the risk of misspeculation with the benefits of further reducing bitwidth.

Seeing the large performance degradation on *MIN*, we investigate further. For all benchmarks, dynamic instructions increase above Baseline by up to 48.6%, 12.5% on average. Knowing that more misspeculations leads to more dynamic instructions executed in CFG_{orig}, we examined the quality of generated assembly in CFG_{orig}, finding it to be significantly worse than Baseline. This is caused by a heuristic used in the register allocator. We inject artificially low branch weights for handlers, implying that they will almost *never* be entered, to prioritize efficient register allocation in CFG_{spec}. We invert this heuristic to imply that handlers will almost *always* be entered. With this change, BitSpec incurs only a 2.6% increase in dynamic instructions over Baseline on average, ranging from a 9.8% increase to a 0.2% reduction. This points to poor register allocation in CFG_{orig}—which sees increased execution due to more misspeculations—as the main cause for increased energy consumption on *MIN*.

RQ6: Sensitivity Study

As a profile-guided technique, it is necessary to examine the input sensitivity of BitSpec. We use alternate inputs for each benchmark from random generators provided with MiBench and well-established inputs from the domain when no generator is provided. We use the alternate input to profile, then run the program with the provided input. Figure 15 shows that BitSpec is robust to alternate inputs for the profiler.

To understand the sensitivity of different selection heuristics, we do a deep dive into `susan-edges` We use a random sample of 50 images from BSDS500 [3] to perform the following experiment: For each image $i \in I$, compile with i as the profile input, producing p_i. For each image $j \in I$, run p_i on j, compute its dynamic instructions relative to p_j run on j. Repeat the experiment for each selection heuristic, i.e., *MAX*, *AVG*, *MIN*. Figure 16 shows the experimental results

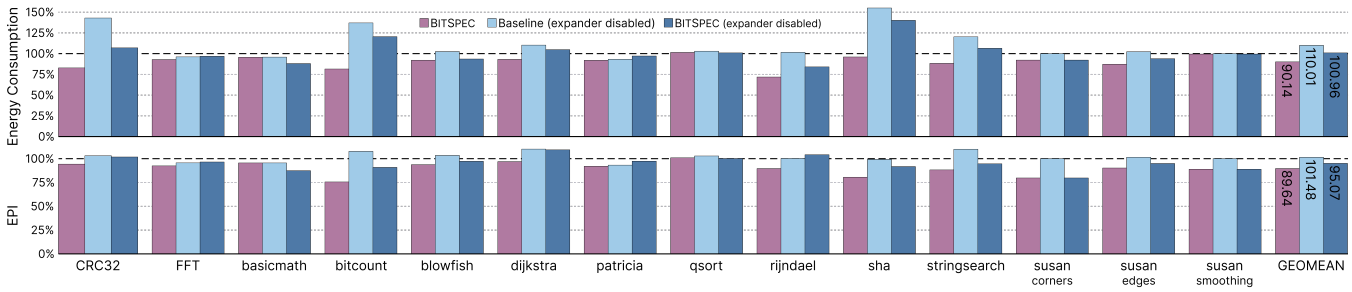

Figure 13. When the expander is disabled, we see that BitSpec does not obtain the same magnitude of energy consumption reduction.

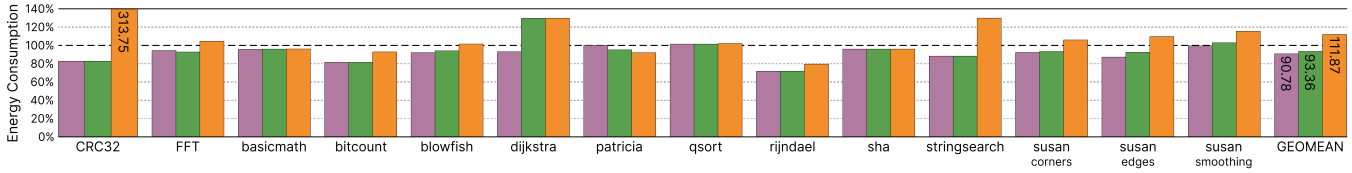

Figure 14. More aggressive bitwidth selection heuristics increase energy consumption, with the exception of `patricia` and `FFT`.

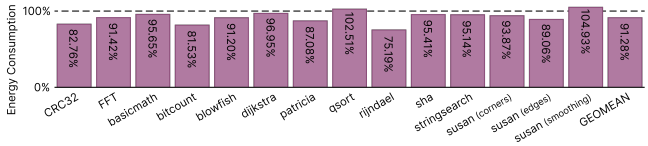

Figure 15. BitSpec is robust to changes in the profiler input, maintaining energy savings with only a 1.14% increase on average.

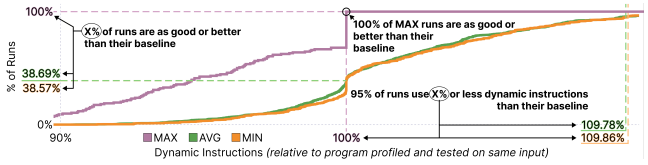

Figure 16. Cumulative distribution of runs for `susan-edges`.

as a cumulative distribution. We conclude that *MAX* generates robust benefits between inputs, while *AVG* and *MIN* are more sensitive. This is caused by *MAX* stabilizing at a shared worst-case, while *AVG* and *MIN* are too aggressive for some image pairs.

RQ7: Fully Automatic Bitwidth Selection

In §2, we identified the gap between programmer-selected and required bitwidth, begging **RQ7**. We modify the original C code of `dijkstra` and `stringsearch`, the only workloads where original bitwidth does not impact correctness, to use 64 bits for all integer variables. If the answer is *yes*, we expect to see the same energy consumption from BitSpec on both the modified and unmodifed programs. For `stringsearch`, BitSpec consumes 11.81% less energy than Baseline running on the unmodified program, while Baseline consumes 16.78% more. For `dijkstra`, BitSpec consumes 0.43% less energy, while Baseline consumes 26.65% more. For `stringsearch`, the answer to **RQ7** is *yes*. However, for `dijkstra` energy consumption is only reduced below Baseline for the modified and unmodified programs, but does not reach parity. Though short of achieving the promise, BitSpec takes a large step towards the goal.

RQ8: Composition with Other Research Architectures

We now examine the application of BitSpec to dynamic timing slack (DTS), which has emerged as a promising target for energy optimization in tiny devices. DTS refers to the portion of the clock period that remains after all signals have propagated through logic paths. Prior work [15] can detect the presence of this slack within a single clock cycle and scale down the supply voltage to reclaim energy.

As ideas, BitSpec and DTS are orthogonal. We aim to see if they compose, as they share an application domain. To do so, we introduce two processors for evaluation. DTS: an implementation of *time squeezing* [16], a DTS-optimized compiler-architecture co-design. DTS+BitSpec: an implementation of *time squeezing*, where the compiler and architecture were extended with BitSpec. The compiler estimates the available DTS in a sequence of instructions and provides a clock period hint, which is used to configure a programmable clock generator—implemented with a multiphase ADPLL—which scales the clock per instruction. To account for changes to the clock period, we scale our energy results with well-established power [31] and delay [37] equations. We use RazorII [11] style error detection and recovery.

Figure 17 shows that DTS and DTS+BitSpec reduce energy consumption by 28.39% and 34.95% on average, respectively. With Figure 8, these results show that the energy savings of DTS+BitSpec is roughly the product of DTS and BitSpec for all benchmarks. Finding that DTS+BitSpec is the sum of

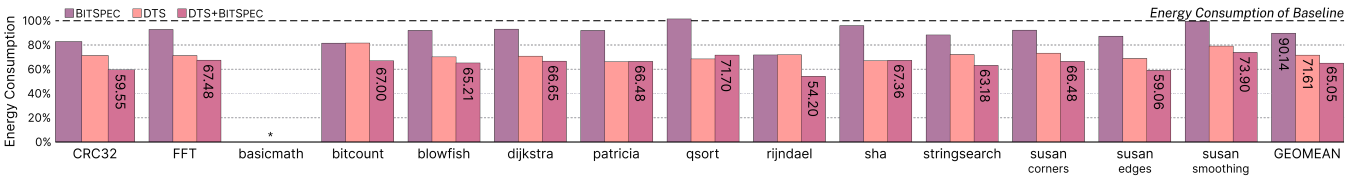

Figure 17. BitSpec combined with a DTS-optimized architecture reduces energy consumption by up to 45.8%, 38.8% on average. `basicmath` is not included due to a compiler bug in the DTS artifact.

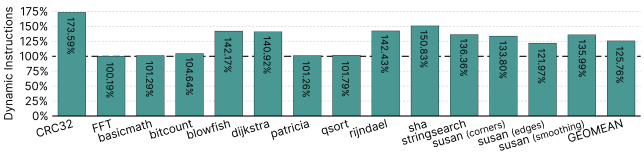

Figure 18. The ARM Thumb architecture requires more dynamic instructions, limiting its applicability to BitSpec.

its parts, we come to two conclusions. First, compiler-based DTS estimation techniques are capable of consistent benefits on both speculative and non-speculative code. Second, these results suggest promising future work in new compiler-based DTS estimation techniques that account for reduced bitwidth execution, which induce shorter critical paths (e.g., ALU carry chains) for DTS-optimized architectures to exploit for further energy savings. Such work would enable DTS+BitSpec to become *more than the sum of its parts*.

RQ9: Other Architectures in Production

Another line of work that has been deployed in production-quality, embedded processors is compact ISA design, such as ARM's Thumb ISA. In our implementation of BitSpec, we extended the ARM 32-bit ISA rather than ARM's compact Thumb ISA. We did not use Thumb because we found that it executes more dynamic instructions than Baseline for our target benchmarks—25.76% on average, up to 73.59%.

5 Related Works and Discussion

Prior work has similarly explored the impact of reduced bitwidth storage and computation, with various approaches to bitwidth selection and register allocation techniques.

Software Approaches to Bitwidth Selection use static analysis to reduce the bitwidth of floating-point [20, 41] and integer [8, 38, 40] variables. However, as seen in Figure 1c, production-quality implementations fall short. Furthermore, static analyses will *never* be able to achieve an optimal bitwidth selection when bitwidth utilization is input-dependent, which BitSpec's profile-guided approach avoids.

Hardware Approaches to Bitwidth Selection reduce the bitwidth of the entire datapath. Pokam et al. [34] leveraged drowsy states [17] in the register file to speculatively reduce datapath bitwidth per basic block. However, this technique is limited by its coarse granularity—as evaluated in Figure 1d—and its blanket treatment of datapath bitwidth, which precludes it from improving register allocation as BitSpec does. Bhagat et al. [5, 6] used a 16-bit ISA, decomposing *all* operations into their 16-bit constituents. With profile-guided speculation techniques, they elide computation on 16-bit slices. Unlike this approach, BitSpec does not require drastic changes to the ISA, only requiring small extensions.

Register allocators have been proposed to pack multiple low-bit variables into a single register [4, 14, 35, 39], but they do not reduce the bitwidth requirements of program variables. Other work speculatively allocates two 32-bit variables into 16-bit slices of a 32-bit register [28]. However, this work does not allow operations on the upper slice of registers, inducing costly data movement avoided by BitSpec.

SIMD units have seen steadily increasing vector widths year after year, with Intel's AVX-512 [36] supporting vectors of up to 512 bits. By applying BitSpec to SIMD instructions, the *effective* vector width is increased, e.g., speculatively transforming 8x64 vectors into 64x8 vectors. But this begs the question: Does speculation on multiple vector elements increase the granularity too much to realize the actual bitwidth utilization? We leave this for future work.

6 Conclusion

In this paper we proposed BitSpec, a compiler-architecture co-design to support per-variable, speculative bitwidth selection, bridging the gap between *programmer-specified* bitwidth and the bitwidth *required* for computation. Our evaluation shows that BitSpec can drastically increase the optimization space presented to register allocators, improving the point of diminishing returns for conventional compiler optimizations, while composing well with prior work on tiny devices. As a framework, BitSpec opens the door to new speculative bitwidth selection techniques, which we believe is key to lifting the burden of manual bitwidth selection.

Acknowledgments

We would like to thank Michael Threatt, Souradip Ghosh and Enrico Deiana for their assistance early in the project, as well as Federico Sossai for his insights on §2.1. We also thank the anonymous reviewers for insightful comments and invaluable suggestions. This material is based upon work supported by the National Science Foundation under NSF-1908488, NSF-2107042, NSF-2119069, NSF-2148177.

References

[1] Frances E. Allen. Control flow analysis. In *Symposium on Compiler Optimization*, 1970. doi: 10.1145/800028.808479. URL https://doi.org/10.1145/800028.808479.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 303–316, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328098. doi: 10.1145/2628071.2628092. URL https://doi.org/10.1145/2628071.2628092.

[3] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, pages 898–916, May 2011. doi: 10.1109/TPAMI.2010.161.

[4] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In George Almási, Călin Caşcaval, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, pages 267–282, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-72521-3.

[5] I. Bhagat, E. Gibert, J. Sanchez, and A. Gonzalez. Eliminating non-productive memory operations in narrow-bitwidth architectures. In *Workshop on Optimizations for DSP and Embedded Systems*, pages 21–29, Apr 2011.

[6] Indu Bhagat, Enric Gibert, Jesús Sánchez, and Antonio González. Global productiveness propagation: A code optimization technique to speculatively prune useless narrow computations. In *LCTES*, 2011. doi: 10.1145/1967677.1967700.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718.

[8] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. Bitvalue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, page 969–979, Berlin, Heidelberg, 2000. Springer-Verlag. ISBN 3540679561.

[9] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160, 2014.

[10] Tung-Chien Chen, Chung-Jr Lian, and Liang-Gee Chen. Hardware architecture design of an h. 264/avc video codec. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 750–757, 2006.

[11] Shidhartha Das, Carlos Tokunaga, Sanjay Pant, Wei-Hsiang Ma, Sudherssen Kalaiselvan, Kevin Lai, David M Bull, and David T Blaauw. Razorii: In situ error detection and correction for pvt and ser tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):32–48, 2008.

[12] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *PLDI*, 2012. doi: 10.1145/2254064.2254120. URL https://doi.org/10.1145/2254064.2254120.

[13] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[14] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 304–315. IEEE, 2004.

[15] Yuanbo Fan, Tianyu Jia, Jie Gu, Simone Campanoni, and Russ Joseph. Compiler-guided instruction-level clock scheduling for timing speculative processors. In *DAC*, 2018. doi: 10.1145/3195970.3196013.

[16] Yuanbo Fan, Simone Campanoni, and Russ Joseph. Time squeezing for tiny devices. In *ISCA*, 2019. doi: 10.1145/3307650.3322268.

[17] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA*, 2002. doi: 10.1145/545214.545232.

[18] LLVM Foundation. Llvm language reference manual — llvm 10 documentation. https://releases.llvm.org/10.0.0/docs/LangRef.html, .

[19] LLVM Foundation. Machine ir (mir) format reference manual. https://releases.llvm.org/10.0.0/docs/MIRLangRef.html, .

[20] A.A. Gaffar, O. Mencer, and W. Luk. Unifying bit-width optimisation for fixed-point and floating-point designs. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 79–88, 2004. doi: 10.1109/FCCM.2004.59.

[21] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, may 1996. ISSN 0164-0925. doi: 10.1145/229542.229546. URL https://doi.org/10.1145/229542.229546.

[22] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. Snafu: An ultra-low-power, energy-minimal cgra-generation framework and architecture. In *ISCA*, 2021. doi: 10.1109/ISCA52012.2021.00084.

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. doi: 10.1109/WWC.2001.15. URL https://doi.org/10.1109/WWC.2001.15.

[24] T. Jia, R. Joseph, and Jie Gu. Greybox design methodology: A program driven hardware co-optimization with ultra-dynamic clock management. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. doi: 10.1145/3061639.3062255.

[25] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. In *PLDI*, 1994. doi: 10.1145/178243.178258. URL https://doi.org/10.1145/178243.178258.

[26] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David M. Brooks. Mallacc: Accelerating memory allocation. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 33–45. ACM, 2017. doi: 10.1145/3037697.3037736. URL https://doi.org/10.1145/3037697.3037736.

[27] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[28] Bengu Li, Youtao Zhang, and Rajiv Gupta. Speculative subword register allocation in embedded processors. In *Languages and Compilers for High Performance Computing*, 2005. doi: 10.1007/11532378_6.

[29] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Comput. Archit. Lett.*, 19(2):106–109, jul 2020. doi: 10.1109/LCA.2020.2973991.

[30] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, Tommy McMichen, David I. August, and Simone Campanoni. Noelle offers empowering llvm extensions. In *CGO*, 2022. doi: 10.1109/CGO53902.2022.9741276.

[31] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[32] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. Domain specialization is generally unnecessary for

accelerators. *IEEE Micro*, 2017. URL https://doi.org/10.1109/MM.2017.60.

[33] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, June 2003. ISSN 0163-5999. doi: 10.1145/885651.781076. URL http://doi.acm.org/10.1145/885651.781076.

[34] Gilles Pokam, Olivier Rochecouste, André Seznec, and François Bodin. Speculative software management of datapath-width for energy optimization. In *LCTES*, 2004. doi: 10.1145/997163.997175.

[35] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2008.

[36] Chris MacNamara Ray Kinsella and Georgii Tkachuk. *Intel AVX-512 - Instruction Set for Packet Processing*. Intel, 2021.

[37] Takayasu Sakurai and A Richard Newton. Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas. *IEEE Journal of solid-state circuits*, 25(2):584–594, 1990.

[38] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI*, 2000. URL https://doi.org/10.1145/349299.349317.

[39] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 85–96, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136285. doi: 10.1145/604131.604139. URL https://doi.org/10.1145/604131.604139.

[40] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *CGO*, 2020. URL https://dl.acm.org/doi/10.1145/3368826.3377927.

[41] J.Y.F. Tong, D. Nagle, and R.A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):273–286, 2000. doi: 10.1109/92.845894.

[42] Shlomo Weiss and James E Smith. A study of scalar compilation techniques for pipelined supercomputers. *ACM SIGARCH Computer Architecture News*, 15(5):105–109, 1987.

[43] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *ISCA*, 2020. doi: https://doi.org/10.1109/ISCA45697.2020.00032.

[44] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA*, 2003. doi: 10.1109/ISCA.2003.1206991.

A Artifact Appendix

A.1 Abstract

Our artifact includes source files for the BitSpec project, including the compiler, functional simulator, gate-level simulator and energy model described and evaluated in the paper.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm: N/A**
- **Program: MiBench benchmark suite (source included)**
- **Compilation: LLVM10, GCC**
- **Transformations: Compiler passes (included)**
- **Data set: MiBench benchmark suite (included)**
- **Run-time environment: Linux**
- **Hardware: Tested on an x64 machine.**
- **Metrics: Dynamic instructions, Energy consumption**
- **Output: Text files**
- **Experiments: Performance and Compiler Evaluation**
- **How much disk space required (approximately)?: Generated artifact is 30 GiB**
- **How much time is needed to prepare workflow (approximately)?: 4 hours**
- **How much time is needed to complete experiments (approximately)?: 2 hours (functional simulation only), >24 hours (gate-level energy)**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT License**
- **Workflow framework used?: See Appendix A.7.**
- **Archived?: https://doi.org/10.5281/zenodo.14776342**

A.3 Description

A.3.1 How Delivered. The artifact is available on Zenodo, at https://zenodo.org/records/14776342.

A.3.2 Hardware Dependencies. Tested on Intel x64 machine.

A.3.3 Software Dependencies. Dependencies installed with Docker, tested using Podman 3.4.4.

A.4 Installation

Download and unpack the artifact tarball from the public repository. Run the following commands from within the installed artifact directory:

```
docker build -t asplos25-artifact -f Dockerfile .
docker run -it asplos25-artifact
```

After installation, run `make` to build the system.

A.5 Experiment Workflow

Experiments can be run from the `automation` directory with:

```
./scripts/run.sh <CONFIG_FILE> <LABEL> <DELTA>
```

`CONFIG_FILE` is a YAML file that specifies how to run the experiment. To run BitSpec with the *MAX* heuristic on all benchmarks use `./configs/mibench/2cfgmax.yml`. To run all benchmarks with the baseline system use `./configs/mibench/baseline.yml`. The `DELTA` value is used to compile and run BitSpec programs, as described in §3.3.4, the default value `4000` works for MiBench programs. Larger programs may need you to specify a larger `DELTA` value. More information about customizing experiments is available in Appendix A.7.

A.6 Evaluation and Expected Results

As the output of the experiment flow, the `output/` directory will be populated with each experiments results. Energy consumption results are reported in `output/<EXPERIMENT>/output/energy.csv`. Dynamic instruction counts are reported in `output/<EXPERIMENT>/output/stats.txt` as `sim_insts`.

A.7 Experiment Customization

Experiments are configured with a YAML file in `automation/configs`. This allows you to select the architecture, benchmark, inputs, compiler and compiler heuristic. The following example is the configuration used to run the CRC32 benchmark with BitSpec using the *MAX* heuristic:

```
CRC32_MAX:
    arch: bitspec
    voltage-scaling: nominal
    isa: ARM_BS_MISSPEC
    benchmark: crc32
    arguments: large.pcm
    compiler: bitwidth_speculation
    middle-end: 2cfg-max
```

Each configuration can be customized as follows. The benchmark can be selected with `benchmark: <NAME>` using its directory name in `benchmark/mibench/<NAME>`. The test and train inputs are set with `arguments: ...` and `train-arguments: ...`.

The architecture can be selected between a baseline ARM processor and the BitSpec processor:

```
arch: {baseline, bitspec}
```

Invoke the compiler for either the baseline ARM ISA or the modified BitSpec ISA:

```
isa: {ARM, ARM_BS_MISSPEC}
```

Set baseline (`baseline_10`), DTS (`time_squeezer_10`), BitSpec (`bitwidth_speculation`) or BitSpec+DTS (`bw_with_ts`) compiler with `compiler: <COMPILER>`. If using a DTS compiler, set `voltage-scaling: timesqueezing`. If using a BitSpec compiler, select the bitwidth selection heuristic:

```
middle-end: {none,2cfg-{max,avg,min}}
```

The expander can be enabled/disabled:

```
expander: {true, false}
```

The `automation/configs` directory has preset configurations.

A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging