*Appears in the Proceedings of the 2016 IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS), April 2016*

# JIT-Assisted Fast-Forward Embedding and Instrumentation to Enable Fast, Accurate, and Agile Simulation

Berkin Ilbeyi and Christopher Batten
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{bi45,cbatten}@cornell.edu

***Abstract*—Computer architects need fast and accurate simulation to research new computing systems, but architects are also increasingly demanding agile simulation to give them flexibility to productively explore the interaction between software and hardware. In this paper, we propose JIT-assisted fast-forward embedding (JIT-FFE) and JIT-assisted fast-forward instrumentation (JIT-FFI) for fast, accurate, and agile simulation. JIT-FFE enables zero-copy architectural state transfer between a state-of-the-art dynamic-binary-translation-based instruction-set simulator and a detailed microarchitectural simulator. JIT-FFI enables productive implementation of fast functional profiling and warmup. We have implemented these two techniques in a new tool, called PydginFF, which can be integrated with any C/C++ detailed simulator. We evaluate PydginFF within the context of the gem5 detailed simulator for both periodic sampling (SMARTS) and targeted sampling (SimPoint) and demonstrate that PydginFF reduces simulation time of fast-forward-based sampling by over 10×.**

## I. Introduction

There is always a need for fast and accurate simulation of computer systems, but achieving these goals is particularly challenging when exploring new architectures for which native functional-first (trace-driven) simulation [14, 20, 30, 31, 33, 42, 51] is not applicable. Traditional approaches to improving the speed and accuracy of simulation for new architectures usually assume that the instruction-set architecture (ISA) and the entire software stack are fixed. For example, checkpoint-based sampling will collect a set of checkpoints from a profiling phase for a specific ISA and software stack, and then reuse these checkpoints across many different experiments. However, there is an emerging trend towards vertically integrated computer architecture research that involves simultaneously rethinking applications, algorithms, compilers, run-times, instruction sets, microarchitecture, and VLSI implementation. Vertically integrated computer architecture research demands *agile simulation*, which allows complete flexibility in terms of exploring the interaction between software and hardware. Agile simulation is certainly possible if one is willing to sacrifice either speed or accuracy. Table I illustrates how the state-of-the-art simulation methodologies can be fast and agile (e.g., instruction-set simulation with dynamic binary translation), accurate and agile (e.g., fast-forward-based sampling), or fast and accurate (e.g., checkpoint-based sampling). A newer trend is to use native execution to speed up functional simulation when the target (i.e., the simulated architecture) and the host (i.e., the architecture running the simulator) are identical. Such "native-on-native" fast-forward acceleration is fast, accurate, and partially agile; this technique enables exploring changes to the software and microarchitecture, but does not enable research that involves changing the actual ISA. Our goal in this paper is to explore a new approach that can potentially enable fast, accurate, and agile simulation for the entire computing stack.

Table I. Simulator Methodologies

| | Fast | Accurate | Agile | Examples |
|---|---|---|---|---|
| DBT-Based ISS | ● | ○ | ● | [1, 9, 28, 29, 32, 34, 45, 57] |
| FF-Based Sampling | ○ | ● | ● | [24, 39, 54, 55, 63, 64] |
| CK-Based Sampling | ● | ● | ○ | [50, 59, 61] |
| NNFF-Based Sampling | ● | ● | ◐ | [4, 6, 36, 51, 52] |
| PydginFF | ● | ● | ● | |

Comparison of different simulator methodologies for achieving fast, accurate, and agile simulation. DBT = dynamic-binary translation; ISS = instruction-set simulation; FF = fast-forward; CK = checkpoint; NNFF = native-on-native fast-forward acceleration.

Simple interpreter-based *instruction-set simulators* (ISSs) have significant overheads in fetching, decoding, and dispatching target instructions. Augmenting an ISS with *dynamic binary translation* (DBT) can enable fast and agile simulation. DBT identifies hot paths through the binary and dynamically translates target instructions on this hot path into host instructions. This eliminates much of the overhead of simple interpreter-based ISSs and enables simulation speeds on the order of hundreds of millions of instructions per second. While traditional DBT-based ISSs are known to be difficult to modify, recent work has explored automatically generating DBT-based ISSs from architectural description languages [5,12,29,47,48]. DBT-based ISSs are fast and agile, but these simulators do not accurately model any microarchitectural details.

*Detailed simulation* uses cycle- or register-transfer-level modeling to improve the accuracy of simulation, but at the expense of slower simulation speeds. Modern microarchitectural simulators run at tens of thousands of instructions per second, which means simulating complete real-world programs is practically infeasible (e.g., simulating a few minutes of wall-clock time can require months of simulation time). Researchers have proposed various *sampling techniques* to make detailed simulation of large programs feasible [16, 54, 55, 63, 64]. These techniques either use statistical sampling or light-weight functional profiling of the overall program to identify representative samples from the full execution. Detailed simulation is only required for the samples, yet these techniques can still maintain accuracy within a high confidence interval. Since the samples are usually a small ratio of the whole program, sampling can drastically reduce the amount of time spent in slow detailed simulation. However, the samples tend to be scattered throughout the entire execution, which raises a new challenge with respect to generating the correct architectural state (e.g., general purpose registers, page tables, physical memory) and potentially microarchitectural state (e.g., caches, branch predictors) to initiate detailed simulation of each sample.

*Fast-forward-based (FF-based) sampling* focuses on providing accurate and agile simulation. An interpreter-based ISS is used to "fast-forward" (FF) the program until the starting point of a sample, at which point the simulator copies the architectural state from the interpreter-based ISS into the detailed simulator [16, 54, 55]. Some FF-based sampling schemes also require *functional warmup* where microarchitectural state is also generated during fast forwarding to ensure accurate detailed simulation of the sample [63, 64]. FF-based sampling significantly improves the simulation speed compared to detailed simulation without sampling, but it is still many orders-of-magnitude slower than DBT-based ISS. The execution time tends to be dominated by the interpreter-based ISS used during fast-forwarding, and as a consequence simulating a few minutes of wall-clock time can still require several days.

*Native-on-native fast-forwarding-based (NNFF-based) sampling* uses native execution instead of an interpreter-based ISS for fast forwarding. These techniques typically use virtualization to keep the host and simulated address spaces separate [4,6,36,51,52]. Because NNFF-based sampling uses much faster native execution for functional simulation, it can achieve fast, accurate, and partially agile simulation. NNFF-based sampling enables quickly changing the microarchitecture and software, but ISAs different than the host cannot run natively. New instructions and experimental ISAs cannot take advantage of native execution for fast-forwarding, making such studies unsuitable for NNFF-based sampling.

*Checkpoint-based sampling* focuses on providing fast and accurate simulation. An ISS is used to save checkpoints of the architectural state at the beginning of each sample [50, 59, 61]. Once these checkpoints are generated for a particular hardware/software interface and software stack, they can be loaded from disk to initiate detailed simulation of the samples while varying microarchitectural configuration parameters. Checkpoint-based sampling improves overall simulation time by replacing the slow FF step with a checkpoint load from disk. However, checkpoint-based sampling requires the hardware/-software interface and software stack to be fixed since regenerating these checkpoints is time consuming. Because checkpoint generation is rare, the tools that profile and generate these checkpoints are usually quite slow; it can take many days to regenerate a set of checkpoints after changing the hardware/-software interface or the software stack.

Section II provides background on DBT-based ISS and sampling-based simulation techniques. We make the key observation that while FF-based sampling is both accurate and agile, its speed suffers from slow FF. This motivates our interest in enabling fast, accurate, and agile simulation by augmenting FF-based sampling with recent research on DBT-based ISSs. However, there are several technical challenges involved in integrating these two techniques. DBT-based ISSs and detailed simulators use very different design patterns (e.g., page-based binary translation vs. object-oriented component modeling), data representations (e.g., low-level flat memory arrays vs. hierarchical memory modeling), and design goals (e.g., performance vs. extensibility). These differences significantly complicate exchanging architectural state between DBT-based ISSs and detailed simulators. Furthermore, instrumenting a DBT-based ISS to enable functional profiling and/or functional warmup can be quite difficult requiring intimate knowledge of the DBT internals.

In Sections III and IV, we propose *JIT-assisted fast-forward embedding* (JIT-FFE) and *JIT-assisted fast-forward instrumentation* (JIT-FFI) to enable fast, accurate, and agile simulation. JIT-FFE and -FFI leverage recent work on the RPython meta-tracing just-in-time compilation (JIT) framework for general-purpose dynamic programming languages [2, 10, 11, 41, 44] and the Pydgin framework for productively generating very fast DBT-based ISSs [29]. JIT-FFE enables embedding a full-featured DBT-based ISS into a detailed simulator, such that the DBT-based ISS can have *zero-copy* access (large data structures do not need to be copied) to the detailed simulator's architectural state. JIT-FFI enables productively instrumenting the DBT-based ISS with just a few lines of high-level RPython code, but results in very fast functional profiling and warmup. We have implemented JIT-FFE and -FFI in a new tool, called PydginFF, which can be integrated into any C/C++ detailed simulator.

Section V evaluates PydginFF within the context of the gem5 detailed simulator [7] and two different sampling techniques (periodic sampling through SMARTS [63,64] and targeted sampling through SimPoint [54]) when running a variety of SPEC CINT2006 benchmarks. PydginFF is able to reduce the simulation time of FF-based sampling by over 10×; simulations that previously took 1–14 days can now be completed in just a few hours.

To our knowledge, this is the first work to propose and demonstrate fast, accurate, and agile simulation through the creative integration of DBT-based ISSs and detailed simulation. Unlike related NNFF-based sampling approaches, our work allows the entire computing stack to be modified in an agile manner. We anticipate this approach would be particularly useful for studying ISA extensions or for exploring radical hardware acceleration techniques to improve the performance of emerging workloads where the software is not static (e.g., just-in-time compilation and optimization techniques). The primary contributions of this work are: (1) we propose JIT-assisted fast-forward embedding to elegantly enable zero-copy architectural state transfer between a DBT-based ISS and a detailed simulator; (2) we propose JIT-assisted fast-forward instrumentation to enable productive implementation of fast functional profiling and warmup; and (3) we evaluate these techniques within the context of PydginFF embedded into gem5 using SMARTS and SimPoint sampling techniques and show compelling performance improvements over traditional sampling with interpreter-based fast-forwarding.

## II. Background

In this section, we provide brief background on DBT-based ISSs and sampling-based detailed simulation, including an overview of the SMARTS and SimPoint methodologies used in our evaluation.

### A. DBT-Based Instruction Set Simulation

Instruction-set simulators (ISSs) facilitate software development for new architectures and the rapid exploration and evaluation of instruction-set extensions. In an interpreter-based ISS,

a dispatch loop fetches and decodes target instructions before dispatching to a function that implements the instruction semantics. Dynamic-binary translation (DBT) can drastically improve the performance of interpreter-based ISSs by removing most of the dispatch-loop-based overheads. A DBT-based ISS still uses an interpreter for light-weight profiling to find frequently executed code regions. These hot regions are then translated into host instruction equivalents. The native assembly code generated using DBT is cached and executed natively whenever possible instead of using the interpreter. DBT-based ISSs require sophisticated software engineering since they include profiling, instruction-level optimizations, assembly code generation, code caching, and a run-time that can easily switch between interpreter- and DBT-based execution. Coordinating all these components while maintaining correctness and high performance makes DBT-based ISSs very hard to implement, maintain, and extend. However, promising recent work has demonstrated sophisticated frameworks that can automatically generate DBT-based ISSs from architecture description languages [38, 46, 60].

At the same time, there has been significant interest in JIT-optimizing interpreters for dynamic programming languages. For example, JavaScript interpreters in web browsers make heavy use of JIT-optimizations to enable highly interactive web content [58]. Another notable JIT-optimizing interpreter is PyPy for the Python language [2, 10, 11, 41, 44]. The PyPy project has created a unique development approach that utilizes the *RPython translation toolchain* to abstract the process of language interpreter design from low-level implementation details and performance optimizations. The interpreter developers write their interpreter (e.g., for the Python language) in a statically typed subset of Python called *RPython*. Using the RPython translation toolchain, an interpreter written in the RPython language is translated into C by going through *type inference*, *back-end optimization*, and *code generation* phases. The translated C code for the interpreter is compiled using a standard C compiler to generate a fast interpreter for the target language. In addition, the interpreter designers can add light-weight *JIT annotations* to the interpreter code (e.g., annotating the interpreter loop, annotating which variables in the interpreter denote the current position in the target program, annotating when the target language executes a backwards branch). Using these annotations, the RPython translation toolchain can automatically insert a JIT into the compiled interpreter binary. RPython separates the language interpreter design from the JIT and other low-level details by using the concept of a *meta-tracing JIT*. In a traditional tracing JIT, a trace of the target language program is JIT compiled and optimized. In a meta-tracing JIT, the trace is generated from the *interpreter* interpreting the target language program. Tracing JITs need to be specifically designed and optimized for each language, while meta-tracing JITs are automatically generated from the annotated language interpreter. The meta-tracing JIT approach removes the need to write a custom JIT compiler for every new language.

Pydgin is a recent framework for productively building DBT-based ISSs [29]. Pydgin makes use of the RPython translation toolchain to bridge the productivity-performance gap between interpreter- and DBT-based ISSs. An ISS in Pydgin is written as an interpreter in the RPython language, which allows it to be translated and compiled into an efficient binary. Pydgin also has the necessary annotations to allow RPython to automatically generate a very fast and optimized JIT. RPython's pseudocode-like syntax and the Pydgin library hide most performance-focused optimizations from the ISA definition, making modifying or adding new instructions very productive. Pydgin is a key enabler for the two techniques proposed in this paper: JIT-assisted fast-forward embedding and JIT-assisted fast-forward instrumentation.

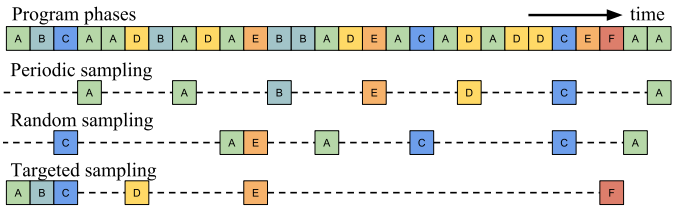


Figure 1. Sampling Methods – Different phases are represented with different letters. Sampling methods include periodic, random, and targeted. The selected representative samples are simulated using detailed simulation; the portion of the execution shown with a dashed line can use fast-forwarding or checkpointing.

### B. Sampling-Based Detailed Simulation

Figure 1 illustrates three approaches to sampling-based detailed simulation: random sampling, periodic sampling, and targeted sampling. Random sampling leverages the central limit theorem to enable calculating confidence bounds on performance estimates. Periodic sampling is an approximation of random sampling, and similar statistical tools can be used to ensure accuracy. Periodicity in the studied programs might skew the results for periodic sampling, however, this was shown not to be an issue for large benchmarks in practice [63,64]. Targeted sampling requires a profiling step to find samples that are representative of different program regions. The profiling information can be microarchitecture-dependent [55] or microarchitecture-independent (e.g., based on basic-block structure [54] or loop/call graphs [26, 27]).

A key challenge in sampling-based detailed simulation is generating the architectural (and potentially microarchitectural) state to initiate the detailed simulation of each sample. Table II shows different types of simulation to produce the initial state for detailed simulation of each sample. *Fast-forwarding* is pure functional execution of the program and only the architectural state is modeled. This is the least detailed type of simulation, hence it tends to be the fastest. However, uninitialized microarchitectural state at the beginning of a sample can heavily bias the results. Researchers usually use some form of warmup to minimize this *cold-start bias*. The processor core pipeline contains relatively little "history" and thus requires warmup of a few thousand instructions. Caches, branch predictors, and transaction look-aside buffers contain much more "history" and thus require hundreds of thousands of instructions to minimize the cold-start bias [17, 19, 63]. *Detailed warmup* will initiate detailed simulation before the start of the sample. Detailed warmup is good for warming up both long- and short-history microarchitecture, but is obviously quite slow. *Func-*

TABLE II. SIMULATION TERMINOLOGY

| | Long-History Modeling | Short-History Modeling | Collect Statistics |
|---|---|---|---|
| Fast-Forwarding | | | |
| Functional Warmup | ✓ | | |
| Functional Profiling | | | ✓ |
| Detailed Warmup | ✓ | ✓ | |
| Detailed Simulation | ✓ | ✓ | ✓ |

Long-history modeling includes caches and branch predictors. Short-history modeling includes core pipeline. Collecting statistics might include profiling or microarchitectural statistics.

TABLE III. COMPARISON OF SMARTS AND SIMPOINT

| | SMARTS | SimPoint |
|---|---|---|
| Sampling Type | periodic | targeted |
| Functional Profiling | optional | required |
| Num of Samples | 10,000 | maximum 30 |
| Len of Samples | 1000 | 10 million |
| Len of Detailed Warmup | 2000 | optional |
| Between Samples | functional warmup | fast forwarding |

SMARTS has an optional profiling step to determine the length of the benchmark; SimPoint has a required functional profiling step to generate BBVs. Length of samples and detailed warmup are in instructions.

*tional warmup* will update long-history microarchitectural state using a purely functional model during fast-forwarding. Because these long-history microarchitectural components tend to be highly regular structures, adding these models to fast-forwarding usually has a modest impact on simulation speed. A related type of simulation is *functional profiling*, which is used in some sampling methodologies to determine where to take samples. Similar to functional warmup, functional simulators can often use light-weight modifications to implement profiling with only a modest impact on simulation speed. Table III compares two common sampling-based simulation methodologies that we will use in our evaluation: SMARTS [61, 63, 64] and SimPoint [26, 27, 39, 54, 59].

*SMARTS* is one of the most well-known statistical sampling methodologies. This approach uses periodic sampling to approximate random sampling, which allows the authors to use statistical sampling theory to calculate confidence intervals for the performance estimates. While the original paper thoroughly evaluates different parameters such as the length of each sample, the number of samples, and the amount of detailed warmup, the paper ultimately prescribes for an 8-way processor: 10,000 samples, each of them 1000 instructions long, with 2000 instructions of detailed warmup [63]. SMARTS is able to use a relatively short length of detailed warmup by relying on functional warmup between samples. The authors determined that if functional warmup is unavailable and pure fast-forwarding is used instead, detailed warmup of more than 500,000 instructions (i.e., 500× the size of the sample) is required for some benchmarks.

*SimPoint* is one of the most well-known targeted sampling methodologies. SimPoint classifies regions of dynamic execution by their *signature* generated from the frequency of basic blocks executed in each region. SimPoint requires an initial functional profiling phase that generates basic block vectors (BBVs) for each simulation interval. Each element of the BBV indicates the number of dynamic instructions executed belonging to a particular basic block. Because the number of basic blocks is large, the dimensionality of the BBVs are reduced using random projection and then classified using the *k*-means clustering algorithm. One simulation interval from each cluster is picked to be a representative sample or *simulation point*. While the original SimPoint paper used 100 million instructions per sample with a maximum of 10 samples [54], a follow-up work used 1 million instructions per sample with a maximum of 300 samples [39]. The most common parameters used in practice tend to be 10 million instructions per sample with a maximum of 30 samples. In contrast to SMARTS, SimPoint uses fewer but longer samples. This results in the inter-sample intervals of billions of instructions and thus SimPoint lends itself to pure fast-forwarding. Since the samples are very long, the effect of cold-start bias is mitigated and warmup is less critical.

## III. JIT-ASSISTED FAST-FORWARD EMBEDDING

One of the biggest obstacles in augmenting FF-based sampling with a DBT-based ISS is coordinating a single execution context for the target application between the ISS and the detailed simulator. The entire architectural state (and microarchitectural state in the case of functional warmup) needs to be communicated between these two simulators. These simulators often represent architectural state differently, using different data structures, in different alignments, and at different granularities. Performance-oriented DBT-based ISSs tend to represent the architectural state in forms that will facilitate high performance, but detailed simulators often choose more extensible approaches that allow running different ISAs and experiments. Another challenge is to ensure consistency of the architectural state for switching. Dirty lines in the modeled caches and uncommitted modifications to the architectural state (e.g., in host registers in DBT-based ISSs) need to be committed before switching. Once the entire architectural state is gathered in a consistent form, another challenge is marshalling this data, using an interprocess communication method such as writing/reading a file/pipe, and finally unmarshalling and reconstructing the data in the new simulator. Performing all of these tasks without incurring excessively long switching times can be quite challenging.

### A. JIT-FFE Proposal

We propose JIT-assisted fast-forward embedding (JIT-FFE) to address this challenge. JIT-FFE enables the fast DBT-based ISS to be dynamically linked to the slow detailed simulator, obviating the need to run two different processes. Since both simulators share the same memory space, large data structures (e.g., the simulated memory for the target) can be directly manipulated by both the DBT-based ISS and the detailed simulator. This removes the need for marshalling, interprocess communication, and unmarshalling, and thus enables *zero-copy* architectural state transfer. JIT-FFE requires both simulators to obey the same conventions when accessing any shared data structures, even at the expense of slightly reduced performance for the DBT-based ISS. JIT-FFE does not require all data structures

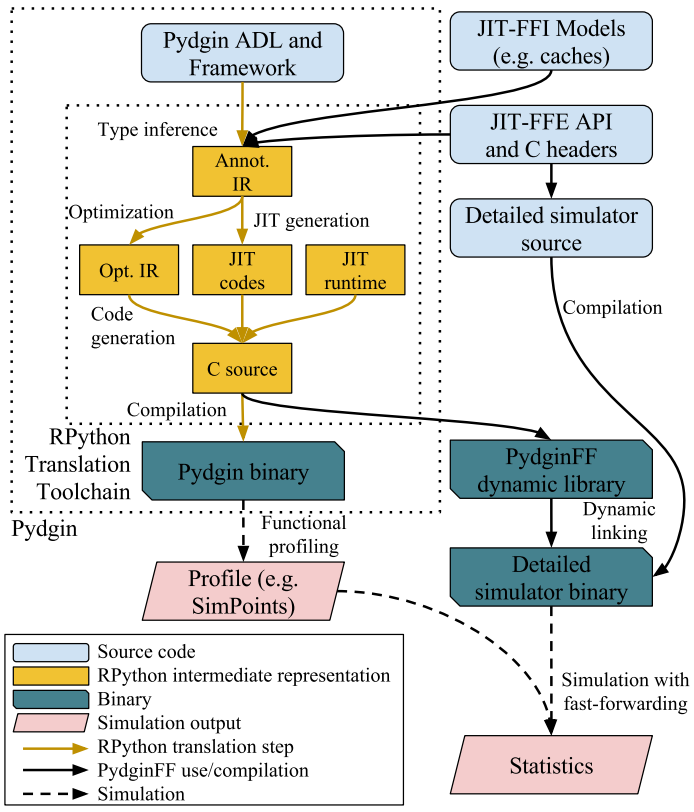


Figure 2. PydginFF compilation and simulation flow.

to be shared; for smaller architectural state (e.g., register file) simply copying the data between simulators is usually simpler. While JIT-FFE elegantly enables using a DBT-based ISS for fast-forwarding in SimPoint, JIT-FFE's ability to provide zero-copy state transfer is particularly effective in SMARTS due to large number of small samples (i.e., many switches between the DBT-based ISS and detailed simulator).

### B. JIT-FFE Implementation

We have augmented Pydgin with JIT-FFE features, which we call PydginFF. PydginFF defines a C/C++ application-programming interface (API) that is visible to the detailed simulator. This API is used for getting and setting the architectural state, declaring a shared data structure for the simulated memory to enable zero-copy state transfer, and starting functional simulation. The API is defined both in the PydginFF source (in the RPython language) and in a C header file. RPython has a rich library to manipulate and use C-language types and structures, and declare C-language entry-point and call-back functions. Even though PydginFF is written in RPython, not C, the RPython translation toolchain translates PydginFF into pure C. This means that after the C/C++ detailed simulator dynamically links against PydginFF, the detailed simulator can directly interact with PydginFF without crossing any language boundaries. This is a key enabler that allows efficient zero-copy architectural state transfer between Pydgin ISS and the detailed simulator.

Figure 2 shows the compilation and simulation flow of PydginFF. Pydgin (without the PydginFF extensions) consists of an architectural description language (ADL) where the ISA is defined, and the framework which provides ISA-independent features and JIT annotations. The ADL and the framework go through the RPython translation toolchain which includes type inference, optimizations, code generation, and JIT generation. This produces the stand-alone JIT-optimized Pydgin binary that can be used for stand-alone functional profiling. The JIT-FFE extensions to Pydgin are also primarily in the RPython language, and go through the same RPython translation toolchain. In addition, we have configured the toolchain to also produce a dynamic library at the end of translation and compilation. C/C++ detailed simulators that are modified to take advantage of the PydginFF API can simply dynamically link against PydginFF and use the Pydgin JIT for fast-forwarding (and functional warmup) in sampled simulation. Targeted sampling methodologies such as SimPoint also need a functional profile of the application. For these, the stand-alone Pydgin binary can generate the functional profile which can be used by the detailed simulator to determine when to start sampling.

## IV. JIT-Assisted Fast-Forward Instrumentation

While JIT-FFE enables a DBT-based ISS to be used for fast-forwarding within a detailed simulator, common sampling methodologies also require instrumenting the functional simulation. For example, SMARTS requires the long-history microarchitectural state to be functionally modelled, and SimPoint uses basic-block vectors generated from functional profiling. Adding instrumentation to a traditional DBT-based ISS can be very challenging. DBT-based ISSs tend to be performance oriented and are not built with extensibility in mind. These simulators are usually written in low-level languages and styles in order to get the best performance. Moreover, the instrumentation code added to these simulators will not automatically benefit from the JIT and can significantly hurt performance.

### A. JIT-FFI Proposal

We propose JIT-assisted fast-forward instrumentation (JIT-FFI) to address this challenge. JIT-FFI allows the researcher to add instrumentation to the RPython interpreter, not to the JIT compiler itself. RPython's meta-tracing JIT approach generates a JIT compiler for the entire interpreter including instruction execution *and* instrumentation. This means instrumentation code is not just inlined but also dynamically JIT-optimized within the context of the target instruction stream. JIT-FFI inlining can produce very high performance for simple instrumentation. However, JIT-FFI inlining can reduce performance if the instrumentation includes complex data-dependent control flow, since this irregularity causes the meta-tracing JIT to frequently abort trace formation. JIT-FFI also includes support for JIT-FFI outlining where the instrumentation code is statically pre-compiled into an optimized function that can be directly called from the JIT trace. Figure 3 shows a simplified example of PydginFF code with JIT-FFI-inlined and -outlined instrumentation code, target instruction stream, and the resulting JIT trace.

JIT-FFI is critical to achieving fast and agile simulation using both SMARTS and SimPoint. For SMARTS, JIT-FFI enables very fast functional warmup of long-history microarchitectural state such as caches. For SimPoint, JIT-FFI enables very fast collection of basic-block vectors for determining representative samples.

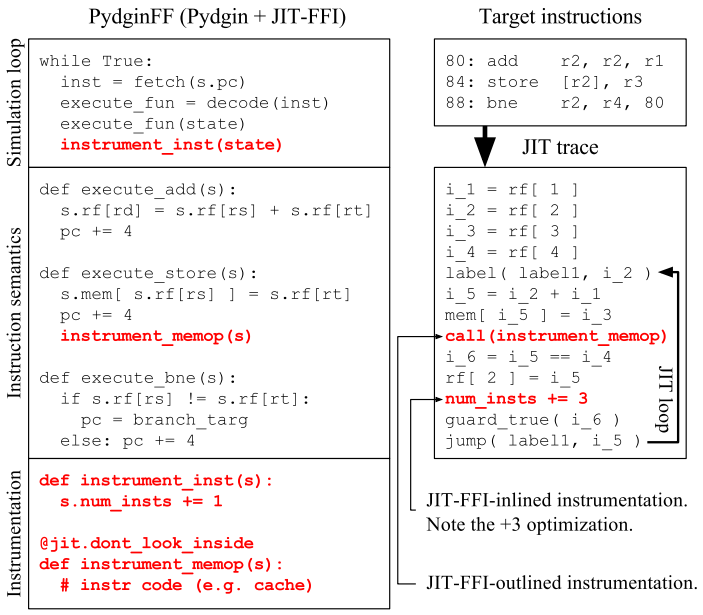


Figure 3. JIT-FFI instrumentation (in **red**) can be added to Pydgin (in black). `instrument_inst` is called from the simulation loop to instrument every instruction; `instrument_memop` is called from the instruction semantics of the store instruction to instrument memory operations. The `@jit.dont_look_inside` decorator causes instrumentation to be JIT-FFI outlined. On the right is a simple loop in an example ISA and the resulting optimized JIT trace with instrumentation.

```
1  def instrument_memop( self, memop_type, address ):
2    line_idx, tag  = self.get_idx_tag( address )
3
4    # get if hit or not, and the way if it was a hit
5    hit, way = self.get_hit_way( line_idx, tag )
6
7    # If miss, get a victim way and update the tag
8    if not hit:
9      way = self.get_victim( line_idx )
10     self.tag_array[ line_idx ][ way ] = tag
11
12   # On write, set dirty bit
13   if memop_type == WRITE:
14     self.dirty_array[ line_idx ][ way ] = True
15
16   # Update LRU bits
17   self.update_lru( line_idx, way )
18
19 @jit.dont_look_inside
20 def get_hit_way( self, line_idx, tag ):
21   for way in range( self.num_ways ):
22     if tag == self.tag_array[ line_idx ][ way ]:
23       return True, way
24   return False, -1
25
26 def update_lru( self, line_idx, way ):
27   self.lru_array[ line_idx ] = 0 if way else 1
28
29 def get_victim( self, line_idx ):
30   return self.lru_array[ line_idx ]
```

Figure 4. JIT-FFI for Cache Warmup – RPython code to model a 2-way set-associative cache. The `@jit.dont_look_inside` decorator can be used to force JIT-FFI outlining.

### B. JIT-FFI Implementation

We illustrate examples of instrumentation that users would write to take advantage of JIT-FFI in functional warmup and functional profiling, respectively. PydginFF provides hooks for users to add custom instrumentation at different granularities (at instruction, memory operation, and control-flow operation levels with `instrument_inst`, `instrument_memop`, `instrument_ctrl` function calls respectively), and more of these hooks can be added. Figure 4 shows the JIT-FFI code within PydginFF to implement a functional model of a 2-way set-associative cache for use in SMARTS. The modeling is done in a rather straight-forward fashion, and the cache model maintains arrays for the tags (`tag_array`), dirty bits (`dirty_array`), and LRU bits (`lru_array`). The RPython language has a rich library of hints that can be given to the JIT, and an example for this can be seen in the `@jit.dont_look_inside` decorator, which leaves the hit lookup function as a call from the JIT trace and forces JIT-FFI outlining. Modeling the cache with a high-level language like RPython and then using simple JIT annotations make JIT-FFI both productive and fast. We evaluate a case study examining the effects of JIT-FFI inlining vs. outlining in set-associative and direct-mapped cache models in Section V-D. Note that it is usually more efficient not to store data in the JIT-FFI caches, but simply retrieve the data from the main memory even on a cache hit (but still update tags and LRU bits accordingly). This means that the cache models do not benefit from JIT-FFE zero-copy architectural state transfer, and their states need to be copied between PydginFF and the detailed simulator.

Figure 5 shows the JIT-FFI code within PydginFF to implement basic-block vector (BBV) generation for use in SimPoint. Basic blocks in BBV generation are defined to be dynamic streams of instructions that have a control-flow instruction at the beginning and end, but not elsewhere. In the `instrument_ctrl` function, line 3 calls `get_bb_idx`, which returns a unique basic-block index or -1 if this basic block has not been seen before. The `@jit.elidable_promote` decorator on line 17 is a JIT annotation that constant-promotes the arguments (guaranteeing that the arguments will always be the same at the same point in the trace), and marks the function elidable (guaranteeing that this function does not have any side effects and always returns the same value). This annotation allows the function to be completely optimized away in the JIT trace and replaced with the corresponding constant basic-block index. In the JIT trace, the basic-block index will never be zero (it would have been observed before), so slower code to register a new basic block (lines 5–9) will be skipped. The only operation that will be inlined in the JIT trace is incrementing the corresponding BBV entry in line 12. This illustrates how JIT-FFI inlining can enable very high-performance instrumentation. We evaluate the effect of enabling JIT-FFI inlining versus outlining for BBV generation in Section V-E.

## V. Evaluation

We have implemented JIT-FFE and -FFI in a new tool, called PydginFF. Although PydginFF can be integrated into any C/C++ detailed simulator, we have chosen the popular gem5

```
# Called only when there are control-flow instructions
def instrument_ctrl( self, old_pc, new_pc, num_insts ):
  bb_idx = self.get_bb_idx( old_pc, new_pc )

  # bb_idx will never be equal to -1 in JIT trace
  if bb_idx == -1:
    # Register a new BB along with the size of the BB
    bb_idx = self.register_new_bb( old_pc, new_pc,
                          num_insts - self.last_num_insts )

  # Increment BBV entry by the size of the bb
  self.bbv[ bb_idx ].increment()
  self.last_num_insts = num_insts

# Get the index into the BBV table
@jit.elidable_promote()
def get_bb_idx( self, old_pc, new_pc ):

  # Construct BB signature, check if BB was seen before
  bb_sig = (old_pc << 32) | new_pc
  if bb_sig not in self.bbv_map:
    return -1

  return self.bbv_map[ bb_sig ]
```

Figure 5. JIT-FFI for BBV Generation – RPython code to generate BBV for SimPoint. `@jit.elidable_promote` is a JIT hint to enable aggressive JIT optimization.

detailed simulator [7] for our evaluation, and we refer to the combined simulator as PydginFF+gem5.

### A. Benchmarks

We quantify the performance of each simulator configuration (baseline SMARTS, baseline SimPoint, PydginFF+gem5 SMARTS, and PydginFF+gem5 SimPoint), using the SPEC CINT2006 benchmark suite. We use the ARMv5 instruction set for our target software, and a Newlib-based GCC 4.3.3 cross-compiler. We use the recommended optimization flags (`-O2`) for compiling the benchmarks. Three of the SPEC benchmarks, *400.perlbench*, *403.gcc*, and *483.xalancbmk* failed to compile for the target configuration due to limited system call support in our cross-compiler so we omit these from our results. Table IV shows the benchmark setup details. We used the *ref* datasets for all of the benchmarks, and picked one dataset in cases where there were multiple datasets. We also show the dynamic instruction counts of the benchmarks, which range from 200 billion to 3 trillion instructions.

### B. Simulation Methodology

Our baseline simulation setup uses the gem5 [7] detailed microarchitecture simulator. We have implemented the SMARTS and SimPoint sampling techniques in gem5. Both baseline configurations use gem5's interpreter-based atomic simple processor for fast-forwarding, functional warmup, and functional profiling; and the cycle-level out-of-order processor model for detailed simulation. We use the ARMv5 instruction set for both PydginFF and gem5 with system-call emulation. For both configurations, the detailed microarchitecture models a 4-way superscalar pipeline with 4 integer ALUs, 2 AGUs, 32 entries in the issue queue, 96 entries in the ROB, and 32 entries each in load and store queues. We model a 2-way 32KB L1 instruction and a 2-way 64KB L1 data cache. The baseline SMARTS and SimPoint implementations use the configurations presented in Table III. The SMARTS configuration only uses functional warmup between the samples, and SimPoint uses fast-forwarding until 500,000 instructions before the start of the simulation point and then switches to detailed warmup. Although optional, it is common practice to use detailed warmup with SimPoint.

TABLE IV. BENCHMARKS

| | | SMARTS | | SimPoint | | |
|---|---|---|---|---|---|---|
| **Benchmark** | **Dataset** | **Dyn Inst** | **% Det Sim** | **Dyn Inst** | **# Spl** | **% Det Sim** |
| 401.bzip2 | `chicken 30` | 195 | 0.020 | 194 | 25 | 0.200 |
| 429.mcf | `inp.in` | 373 | 0.008 | 373 | 27 | 0.100 |
| 445.gobmk | `13x13.tst` | 323 | 0.009 | 316 | 20 | 0.090 |
| 456.hmmer | `nph3 swiss41` | 1112 | 0.003 | 952 | 13 | 0.020 |
| 458.sjeng | `ref.txt` | 2974 | 0.001 | 2921 | 13 | 0.007 |
| 462.libquantum | `1397 8` | 3069 | 0.001 | 3036 | 17 | 0.008 |
| 464.h264ref | `foreman_ref` | 753 | 0.004 | 707 | 15 | 0.030 |
| 471.omnetpp | `omnetpp.ini` | 1282 | 0.002 | 1254 | 3 | 0.004 |
| 473.astar | `BigLakes2048` | 434 | 0.007 | 397 | 15 | 0.060 |

Dyn Inst = number of dynamic instructions when run to completion (in billions); % Det Sim = percentage simulated using the detailed simulator; # Spl = number of samples. SimPoint has slightly fewer dynamic instructions since the simulation can stop after the final sample.

As mentioned in Sections III and IV, we have used the open-source Pydgin DBT-based ISS [29,43] to develop PydginFF. We embed PydginFF into gem5 (PydginFF+gem5) and evaluate the performance, again using SMARTS and SimPoint, against the baseline SMARTS and SimPoint implementations with gem5 alone. The PydginFF+gem5 configurations use the DBT-based ISS for fast-forwarding, functional warmup, and functional profiling, and the same gem5 cycle-level microarchitectural model for detailed warmup and detailed simulation. We use PyPy/R-Python 2.5.1 to translate PydginFF, GCC 4.4.7 to compile both PydginFF and gem5, and SimPoint 3.2 to generate simulation points from BBVs. We run all experiments on an unloaded 4-core 2.40 GHz Intel Xeon E5620 machine with 48 GB of RAM.

Table IV shows sampling-related statistics of the benchmarks we used. A difference between the two sampling techniques is that SimPoint requires running the target program until the end of the last simulation point instead of running it to completion. However, it can be seen that the last simulation points tend to be close to the completion of the program, so this benefit is minimal. The number of simulation points in SimPoint can be varied as well, and our benchmarks showed a large range from 3 to 27. This number determines how much total detailed simulation needs to take place, which can affect the simulation performance. Since SMARTS uses the same number of samples for each benchmark, the total detailed simulation is the same. The table also shows the percentage of detailed simulation (including detailed warmup) that takes place in each sampling technique. These values are extremely low, indicating that fast-forwarding does indeed constitute 99+% of the simulation and motivating the need for faster fast-forwarding and functional warmup.

TABLE V. SMARTS AND SIMPOINT RESULTS

| | gem5 fun | | gem5 det | | Pydgin | | gem5 SM | | gem5 SP | | PydginFF+gem5 SM | | | PydginFF+gem5 SP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **IPS** | **T*** | **IPS** | **T**** | **IPS** | **T** | **IPS** | **T*** | **IPS** | **T*** | **IPS** | **T** | **vs. g5** | **IPS** | **T** | **vs. g5** |
| 401.bzip2 | 2.4M | 23h | 54K | 41d | 613M | 5.3m | 2.2M | 1.0d | 2.0M | 1.1d | 44M | 1.2h | 20× | 29M | 1.9h | 14× |
| 429.mcf | 2.4M | 1.8d | 60K | 72d | 487M | 13m | 2.0M | 2.1d | 1.9M | 2.2d | 34M | 3.1h | 17× | 25M | 4.2h | 13× |
| 445.gobmk | 2.3M | 1.6d | 51K | 74d | 119M | 45m | 2.0M | 1.8d | 1.9M | 1.9d | 14M | 6.3h | 7× | 40M | 2.2h | 20× |
| 456.hmmer | 2.3M | 5.6d | 67K | 192d | 582M | 32m | 2.1M | 6.2d | 1.9M | 5.8d | 49M | 6.4h | 24× | 195M | 1.4h | 102× |
| 458.sjeng | 2.4M | 14d | 58K | 596d | 260M | 3.2h | 2.1M | 16d | 2.1M | 16d | 24M | 1.5d | 11× | 160M | 5.1h | 76× |
| 462.libquantum | 2.6M | 14d | 66K | 534d | 605M | 1.4h | 2.2M | 16d | 2.1M | 17d | 93M | 9.1h | 43× | 292M | 2.9h | 141× |
| 464.h264ref | 2.4M | 3.6d | 66K | 133d | 732M | 17m | 2.1M | 4.1d | 2.0M | 4.0d | 34M | 6.2h | 16× | 157M | 1.2h | 77× |
| 471.omnetpp | 2.8M | 5.4d | 62K | 240d | 474M | 45m | 2.3M | 6.4d | 2.3M | 6.3d | 27M | 13h | 12× | 209M | 1.7h | 90× |
| 473.astar | 2.5M | 2.0d | 64K | 78d | 386M | 19m | 2.1M | 2.3d | 2.0M | 2.3d | 31M | 3.9h | 15× | 67M | 1.6h | 34× |

IPS = inst/second; T = simulation time (T* = extrapolated from 10B inst, T** = extrapolated from 500M inst); vs. g5 = speedup relative to gem5 baseline; fun = pure functional simulation; det = pure detailed simulation; SM = sampling-based simulation with SMARTS, SP = sampling-based simulation with SimPoint.

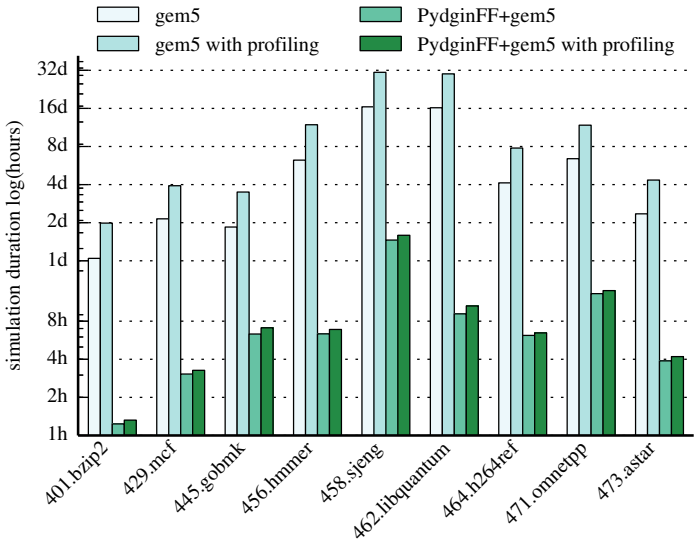


Figure 6. SMARTS Results – gem5 and PydginFF+gem5 results using SMARTS. Note the log scale for duration. with profiling = functional profiling added on top of simulation time.

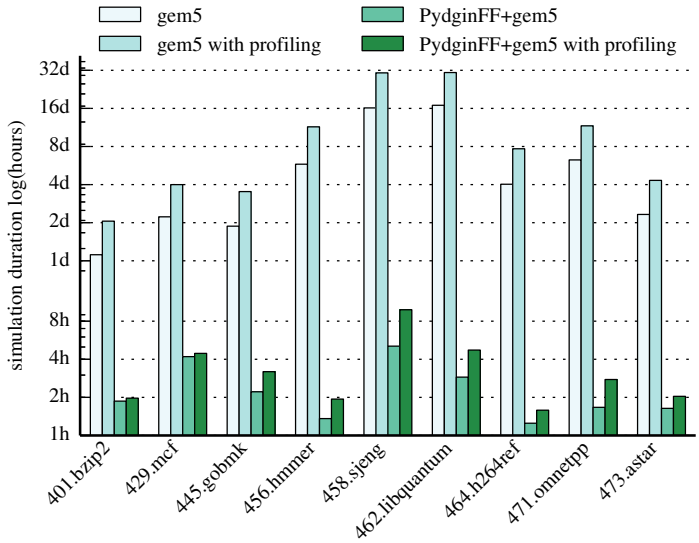


Figure 7. SimPoint Results – gem5 and PydginFF+gem5 results using SimPoint. Note the log scale for duration. with profiling = functional profiling added on top of simulation time.

### C. Overall Results

Table V shows the instructions per second (IPS) and elapsed time results of pure gem5, pure Pydgin, and PydginFF+gem5 configurations. gem5 simulator performance is very consistent across different benchmarks: around 2.5 MIPS and 60 KIPS for functional and detailed simulations, respectively. Even on the functional simulator, the simulations can take many days. Simulating the entire lengths of the longer benchmarks in the detailed model would take well over a year, which clearly is not feasible. Note that the table shows the results for gem5 functional simulation without any cache modeling. However, we also ran gem5 functional simulation with cache modeling (e.g., for functional warmup) and the results were very close to without caches. The Pydgin column shows the performance that a pure DBT-based ISS can achieve, between 100–700 MIPS. It should be noted that DBT-based ISS performance is more varied compared to interpreter-based ISSs like the gem5 atomic model. This variation is due to dynamic optimizations done on instruction streams: some instruction streams benefit more than others. The longest-running benchmark took only about 3 hours on the DBT-based ISS.

The SMARTS results can be seen in Table V (*gem5 SM* and *PydginFF+gem5 SM* columns) and in Figure 6. The plot shows the total duration of simulation in log timescale. In the baseline gem5 SMARTS configuration, simulations can take 1–16 days, similar to gem5 functional model. This is because the vast majority of instructions are being executed in the functional warmup mode. To guarantee low error bounds, SMARTS needs a sufficient number of samples. A functional profiling step hence might be necessary to determine the total number of instructions, which can then be used to compute the distance between samples to reach the target number of samples. Adding this optional functional profiling phase roughly doubles the simulation time on gem5. PydginFF+gem5 using SMARTS sampling performs much better: between one hour to less than two days. The speedup of this technique compared to gem5 can be seen in Table V, which is well over an order of magnitude for most of the benchmarks.

The SimPoint results are also shown in Table V (*gem5 SP* and *PydginFF+gem5 SP* columns) and in Figure 7. The baseline gem5 SimPoint results are similar to SMARTS because the execution time again is dominated by the highly predictable per-

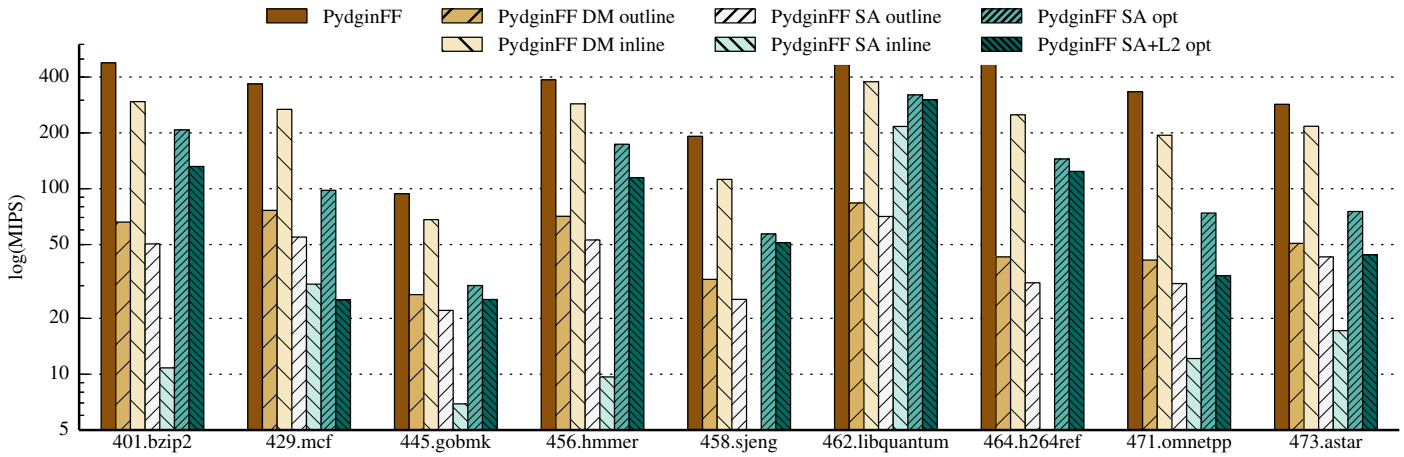


Figure 8. PydginFF Cache Modeling – PydginFF = PydginFF with no cache modeling; PydginFF DM/SA outline/inline = PydginFF modeling a direct-mapped/set-associative cache modeling with JIT-FFI outlining/inlining; SA opt = set-associative cache with optimized JIT-FFI inlining; SA+L2 opt = L1 and L2 caches (both set-associative) with optimized JIT-FFI inlining. Note that this study uses stand-alone PydginFF with no gem5 detailed simulation. 458.sjeng and 464.h264ref with a JIT-FFI inlined SA cache model were aborted due to using too much memory.

formance of the gem5 functional model. PydginFF+gem5 manages to get even better speedups compared to SMARTS (in the range 13–141×) with a maximum execution time of only about five hours. Even including functional profiling, which SimPoint requires for every new binary, PydginFF+gem5 only takes about nine hours in the worst case, compared to 32 days using the baseline. The reason why SimPoint performance is better on PydginFF+gem5 compared to SMARTS is because SimPoint uses fast-forwarding instead of functional warmup, and fast-forwarding on our DBT-based ISS is much faster, as will be shown in Section V-D. One final thing to note is that longer-running benchmarks (e.g., 462.libquantum and 458.sjeng) can get much better speedups compared to shorter-running benchmarks (e.g., 401.bzip2 and 429.mcf) due to the lower ratio of detailed simulation to functional simulation.

### *D. JIT-FFI Case Study: Cache Modeling*

The SMARTS technique requires functional warmup to be used instead of fast-forwarding to generate the microarchitectural state before the detailed simulation starts. Functional warmup requires the long-history microarchitectural structures (e.g., caches) to be simulated in the functional model. We used JIT-FFI to implement direct-mapped and set-associative caches in PydginFF. In Figure 8, we compare the performances of stand-alone PydginFF (without gem5 detailed simulation) using direct-mapped and set-associative cache modeling to PydginFF without caches. We also compare the effects of using JIT-FFI inlining and outlining when implementing the cache model. In this study, PydginFF without caches models virtual memory, so its performance is slightly lower than unmodified Pydgin that uses raw memory. A JIT-FFI outlined direct-mapped cache significantly reduces performance compared to PydginFF without caches (up to 10×). However, JIT-FFI inlining can bring the performance back to reasonable levels (within 50% of PydginFF without caches). Outlined set-associative cache unsurprisingly has an even worse performance than outlined direct-mapped cache due to the increased complexity in the cache model. However, unlike the direct-mapped cache where JIT-FFI inlining helps, JIT-FFI-inlined set-associative cache performs worse that outlining.

The reason for this slowdown is because the set-associative cache model has data-dependent control-flow at line 22 in Figure 4. This if statement checks each cache way for the tag, and this information is recorded in the generated trace. Any time the same static instruction loads from or stores to a memory address that belongs to another cache way, the trace is aborted, which causes the observed speed degradation. These frequent trace aborts usually cause new traces to be generated, each of them with a different control-flow path. For certain benchmarks, the explosion of compiled traces can use up all of the host memory, as was observed in 458.sjeng and 464.h264ref. However, in one benchmark, 462.libquantum, the static loads and store instructions usually hit the same way, and do not display this pathological behavior. Due to these observations, PydginFF uses JIT-FFI outlining for set-associative cache models including for the results in the previous section.

We have also implemented a more optimized set-associative cache model using JIT-FFI inlining (shown as PydginFF SA opt in Figure 8). This uses a relatively large (~1 GB) lookup table for the entire address space to map every cache line to its corresponding way in the cache, or an invalid value to indicate a miss. This removes the data-dependent control flow, and manages to increase the performance to 50–310 MIPS for most benchmarks. Note that for a 64-bit ISA, this optimized implementation might require a prohibitively large lookup table. Figure 8 also shows the impact of adding a set-associative L2 cache; PydginFF is still able to achieve simulation performance between 30–300 MIPS for most benchmarks.

### *E. JIT-FFI Case Study: BBV Generation*

Basic-block vectors (BBVs) generated from the functional profiling step are crucial for the SimPoint sampling methodology. Because these BBVs need to be generated any time the target software changes, it is important that the BBV generation is fast and does not prevent agile simulation. We used JIT-FFI to implement the BBV generation as explained in Section IV-B. We compared unmodified Pydgin to BBV generation with JIT-

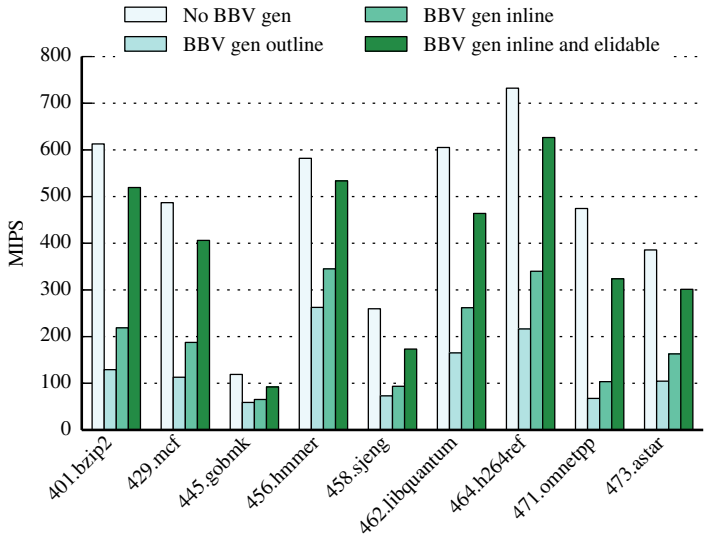


Figure 9. BBV Generation with Optimizations – No BBV gen = unmodified Pydgin that does not generate BBVs; BBV gen outline/inline = BBV generation with JIT-FFI outlining/inlining; elidable = uses the `@jit.elidable_promote` hint. Note that this study uses stand-alone Pydgin and PydginFF with no gem5 detailed simulation.

FFI outlining, JIT-FFI inlining, and using the more advanced `@jit.elidable_promote` hint. These experiments were done on stand-alone Pydgin and PydginFF without gem5 detailed simulation. Figure 9 shows the results for this case study. The goal with inlining and eliding optimizations is to approach the unmodified Pydgin performance, hence have the least possible overhead in generating the BBVs. Figure 9 shows that unoptimized (outlined) BBV generation can have more than 7× slowdown compared to unmodified Pydgin. With the addition of simple JIT annotations, the BBV generation overhead shrinks to around 20%.

## VI. Related Work

There has been significant prior work on DBT-based instruction-set simulators. Seminal work on DBT techniques provided significant performance benefits over traditional interpretive simulation [15, 32, 34, 62]. These performance benefits have been further enhanced by optimizations which reduce overheads and improve code generation quality of just-in-time compilation (JIT) [21, 28, 57]. Current state-of-the-art ISSs incorporate parallel JIT-compilation task farms [9], multicore simulation with JIT-compilation [1, 45], or both [23]. This work builds on top of Pydgin [29], which uses a slightly different approach for building DBT-based simulators. Instead of a custom JIT, Pydgin leverages the RPython framework [2, 10, 11, 41, 44], which allows encoding the architecture description in pure Python. The RPython framework then adds a JIT to the interpreter and translates it to C. There has also been prior work on embedding the cycle-approximate hardware model directly in the JIT [8, 18]. While these approaches tend to be fast and agile for changing the software, they are not agile for changing the modeled microarchitecture.

To reduce simulation time, KleinOsowski et. al proposed manually crafting smaller datasets for the SPEC 2000 suite that were representative of the larger datasets [22]. However, manually manipulating the datasets is not agile for frequent changes in the software stack. Sampling approaches, in contrast, are automated, so they are much more agile in the face of changing software. Researchers proposed picking samples randomly [16, 25], periodically [63, 64], and targeted [24, 54, 55]. Random and periodic sampling methodologies can be used to prove error bounds. Targeted sampling uses either microarchitectural [55] or microarchitecture-independent [24, 54] metrics, to identify samples that are representative of the overall execution. A notable work on SimPoint in the context of modeling different ISAs is Perelman et. al, where the authors propose techniques to synchronize the simulation points on different ISAs to correspond to the same program phases [40]. This work is orthogonal to PydginFF.

Checkpointing is a very popular technique to reduce the simulation time further. Architectural state is usually very large, so checkpoint sizes on the disk is a concern, especially when there are many checkpoints [59]. However, there are proposals to reduce the checkpoint size by analyzing which memory addresses are actually used in each sample, and then only storing these values in the checkpoint [59, 61], and over time cheaper disk space made this issue less of a concern. The most serious shortcoming of checkpointing is its lack of agility: every time the software or the program inputs change, checkpoints need to be re-generated.

Other techniques to improve the simulation time includes Perelman et. al, which proposes an improved version of the SimPoint algorithm where simulation points that occur early in the program execution are prioritized to reduce the amount of fast-forwarding necessary [39]. This technique can be used in conjunction with our proposal to improve simulation time. Wisconsin Wind Tunnel is one of the earliest uses of direct execution to speed up simulator performance [35, 49]. Patil et. al use the Pin [30, 42] dynamic binary instrumentation tool to generate the profiling information to be used by SimPoint [37]. Similarly, Szwed et. al propose native execution for fast-forwarding, by re-compiling the original code to explicitly extract the architectural state to be copied to a detailed simulator for sampling [56]. However, both of these approaches only work if the target ISA is the same as the host ISA, which is usually not the case for new ISA research or studying new ISA extensions. Schnarr and Larus proposed using native execution coupled with microarchitectural modeling of an out-of-order processor [53]. They use memoization to cache timing outcomes of previously seen basic blocks and starting microarchitectural states and skip detailed simulation if they hit in the timing cache. Brankovic et. al looked into the problem of simulating hardware/software co-designed processors which usually include a software optimization layer [13]. For accurate simulation, the software optimization layer needs to be warmed up in addition to the microarchitecture. The authors detect ways to determine when the optimization layer is warmed up, and PydginFF would be an orthogonal technique to speed up the software-optimization-layer warmup.

Another notable work to make sampling-based simulation fast is Full Speed Ahead (FSA) [52] where the authors also acknowledge the need for agility in the context of hardware/-software co-design. FSA uses virtualized native execution for

fast-forwarding until the samples and also uses zero-copy memory transfer when switching between the virtualized native execution and detailed model. However, because FSA relies on native execution, it is not suitable when the ISA is different than the host in the context of ISA extensions or modeling new ISAs. Furthermore, fast functional warmup is not supported on FSA, which is required by SMARTS. The COTSon [4] project uses fast-forward-based sampling, where the SimNow ISS [6] is used to fast-forward between the samples. SimNow uses DBT techniques to speed up the simulation. However, SimNow is not open-source and only supports the x86 ISA, so it is not possible to use this flow for agile ISA extension developments. Other recent attempts to make simulation fast (e.g., Graphite [33], CMP-Sim [20], ZSim [51], MARSS [36], and Sniper [14]) also rely on dynamic binary instrumentation, so are not suitable when the target ISA is different than the host. The only other simulator that uses JIT technology for fast-forwarding that we are aware of is ESESC [3]. ESESC uses JIT-optimized QEMU ISS for fast-forwarding and functional warmup. However, QEMU is a performance-focused ISS that sacrifices productivity for performance, which makes experimenting with ISA extensions challenging. Pydgin focuses both on productivity and performance, and gives researchers agility in modifying the entire computation stack.

## VII. Conclusion

State-of-the-art simulation methodologies can be fast and agile (e.g., instruction-set simulation with dynamic binary translation), accurate and agile (e.g., fast-forward-based sampling), or fast and accurate (e.g., checkpoint-based sampling). Native-on-native fast-forward-based sampling is fast, accurate, but partially agile. In this paper, we have proposed JIT-assisted fast-forward embedding (JIT-FFE) and JIT-assisted fast-forward instrumentation (JIT-FFI) that elegantly enable augmenting a detailed simulator with a DBT-based ISS. We have implemented JIT-FFE and -FFI in a new tool, called PydginFF, and we have evaluated our approach within the context of the gem5 detailed simulator and two different sampling techniques (periodic sampling with SMARTS and targeted sampling with SimPoint). Our results show that PydginFF is able to reduce the simulation time of fast-forward-based sampling by over 10×, truly enabling fast, accurate, and agile simulation.

PydginFF opens up a number of interesting directions for future research. JIT-FFE's ability to provide zero-copy state transfer and JIT-FFI's ability to easily model microarchitectural components can be applied to branch predictors and TLBs. PydginFF can be integrated with any C/C++ microarchitectural simulator, so we are exploring integration with simulators that support a functional/timing split or even register-transfer-level models. Finally, PydginFF enables new kinds of vertically integrated research. For example, PydginFF can enable exploring hardware acceleration and ISA specialization for emerging workloads such as JIT-optimized interpreters of dynamic programming languages, using fast, accurate, and agile simulation. PydginFF will be released as an open-source project.

## Acknowledgments

This work was supported in part by NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, a DARPA Young Faculty Award, and a donation from Intel Corporation. We would also like to thank Trevor Carlson for his insightful feedback about the work.

## References

[1] O. Almer et al. Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2011.

[2] D. Ancona et al. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. *Symp. on Dynamic Languages*, Oct 2007.

[3] E. K. Ardestani and J. Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[4] E. Argollo et al. COTSon: Infrastructure for Full System Simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, Jan 2009.

[5] R. Azevedo et al. The ArchC Architecture Description Language and Tools. *Int'l Journal of Parallel Programming (IJPP)*, 33(5):453–484, Oct 2005.

[6] R. Bedicheck. SimNow: Fast Platform Simulation Purely in Software. *Symp. on High Performance Chips (Hot Chips)*, Aug 2004.

[7] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[8] I. Böhm, B. Franke, and N. Topham. Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2010.

[9] I. Böhm, B. Franke, and N. Topham. Generalized Just-In-Time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2011.

[10] C. F. Bolz et al. Allocation Removal by Partial Evaluation in a Tracing JIT. *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, Jan 2011.

[11] C. F. Bolz et al. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, Jul 2009.

[12] F. Brandner et al. Fast and Accurate Simulation using the LLVM Compiler Framework. *Workshop on Rapid Simulation and Performance Evalution: Methods and Tools (RAPIDO)*, Jan 2009.

[13] A. Branković et al. Warm-Up Simulation Methodology for HW/SW Co-Designed Processors. *Int'l Symp. on Code Generation and Optimization (CGO)*, Feb 2014.

[14] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2011.

[15] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1994.

[16] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing State Loss for Effective Trace Sampling of Superscalar Processors. *Int'l Conf. on Computer Design (ICCD)*, Oct 1996.

[17] L. Eeckhout et al. BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation. *The Computer Journal*, May 2005.

[18] B. Franke. Fast Cycle-Approximate Instruction Set Simulation. *Workshop on Software & Compilers for Embedded Systems (SCOPES)*, Mar 2008.

[19] J. W. Haskins Jr. and K. Skadron. Accelerated Warmup for Sampled Microarchitecture Simulation. *ACM Trans. on Architecture and Code Optimization (TACO)*, Mar 2005.

[20] A. Jaleel et al. A Pin-Based On-the-Fly Multi-Core Cache Simulator. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2008.

[21] D. Jones and N. Topham. High Speed CPU Simulation Using LTU Dynamic Binary Translation. *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, Jan 2009.

[22] A. KleinOsowski et al. Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research. *Int'l Conf. on Computer Design (ICCD)*, Sep 2000.

[23] S. Kyle et al. Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-Based Just-In-Time Dynamic Binary Translation. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2012.

[24] T. Lafage and A. Seznec. Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream. *Workshop on Workload Characterization (WWC)*, Sep 2000.

[25] S. Laha, J. H. Patel, and R. K. Patel. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. on Computers (TC)*, Nov 1988.

[26] J. Lau, E. Perelman, and B. Calder. Selecting Software Phase Markers with Code Structure Analysis. *Int'l Symp. on Code Generation and Optimization (CGO)*, Mar 2006.

[27] J. Lau, S. Schoenmackers, and B. Calder. Structures for Phase Classification. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2004.

[28] Y. Lifshitz et al. Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration. *Workshop on Infrastructures for Software/Hardware Co-Design (WISH)*, Apr 2011.

[29] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2015.

[30] C.-K. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2005.

[31] M. M. K. Martin et al. Multifacet's General Execution Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News (CAN)*, 33(4):92–99, Sep 2005.

[32] C. May. Mimic: A Fast System/370 Simulator. *ACM Sigplan Symp. on Interpreters and Interpretive Techniques*, Jun 1987.

[33] J. E. Miller et al. Graphite: A Distributed Parallel Simulator for Multicores. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan 2010.

[34] W. S. Mong and J. Zhu. DynamoSim: A Trace-Based Dynamically Compiled Instruction Set Simulator. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2004.

[35] S. S. Mukherjee et al. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 1(4):12–20, Oct 2000.

[36] A. Patel et al. MARSS: A Full System Simulator for Multicore x86 CPUs. *Design Automation Conf. (DAC)*, Jun 2011.

[37] H. Patil et al. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[38] D. A. Penry and K. D. Cahill. ADL-Based Specification of Implementation Styles for Functional Simulators. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2011.

[39] E. Perelman, G. Hamerly, and B. Calder. Picking Statisticlally Valid and Early Simulation Points. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2003.

[40] E. Perelman et al. Cross Binary Simulation Points. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2007.

[41] B. Peterson. PyPy. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications, Volume II*. LuLu.com, 2008.

[42] Pin–A Dynamic Binary Instrumentation Tool. Online Webpage, 2012 (accessed Sep, 2015). `http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[43] Pydgin Repository on GitHub. Online Webpage, 2015 (accessed Sep 15, 2015). `http://www.github.com/cornell-brg/pydgin`.

[44] PyPy. Online Webpage, 2014 (accessed Sep 26, 2014). `http://www.pypy.org`.

[45] W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-Compiled Instruction-Set Simulation. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, Oct 2006.

[46] W. Qin and S. Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. *Design, Automation, and Test in Europe (DATE)*, Jun 2003.

[47] W. Qin and S. Malik. A Study of Architecture Description Languages from a Model-based Perspective. *Workshop on Microprocessor Test and Verification (MTV)*, Nov 2005.

[48] W. Qin, S. Rajagopalan, and S. Malik. A Formal Concurrency Model Based Architecture Description Language for Synthesis of Software Development Tools. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2004.

[49] S. K. Reinhardt et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1993.

[50] J. Ringenberg et al. Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2005.

[51] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[52] A. Sandberg et al. Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2015.

[53] E. Schnarr and J. R. Larus. Fast Out-of-Order Processor Simulation Using Memoization. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 1998.

[54] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb 2002.

[55] K. Skadron et al. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques. *IEEE Trans. on Computers (TC)*, Nov 1999.

[56] P. K. Szwed et al. SimSnap: Fast-Forwarding via Native Execution and Application-Level Checkpointing. *Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, Feb 2004.

[57] N. Topham and D. Jones. High Speed CPU Simulation using JIT Binary Translation. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2007.

[58] V8 JavaScript Engine. `https://code.google.com/p/v8`.

[59] M. Van Biesbrouck, B. Calder, and L. Eeckhout. Efficient Sampling Startup for SimPoint. *IEEE Micro*, Jul 2006.

[60] H. Wagstaff et al. Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated from a High-Level Architecture Description. *Design Automation Conf. (DAC)*, Jun 2013.

[61] T. F. Wenisch et al. Simulation Sampling with Live-Points. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2006.

[62] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1996.

[63] R. E. Wunderlich et al. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2003.

[64] R. E. Wunderlich et al. Statistical Sampling of Microarchitecture Simulation. *ACM Trans. on Modeling and Computer Simulation*, Jul 2006.