The Potential of Dynamic Binary Modification and CPU-FPGA SoCs for Simulation

DOI:
10.1109/FCCM.2017.36

Document Version
Accepted author manuscript

Published in:
The 25th IEEE International Symposium on Field-Programmable Custom Computing Machines

The Potential of Dynamic Binary Modification and CPU-FPGA SoCs for Simulation

John Mawer, Oscar Palomar, Cosmin Gorgovan, Andy Nisbet, Will Toms and Mikel Luján
School of Computer Science, University of Manchester, UK
{first}.{last}@manchester.ac.uk

***Abstract*—In this paper we describe a flexible infrastructure that can directly interface unmodified application executables with FPGA hardware acceleration IP in order to 1), facilitate faster computer architecture simulation, and 2), to prototype microarchitecture or accelerator IP. Dynamic binary modification tool plugins are directly interfaced to the application under evaluation via flexible software interfaces provided by a userspace hardware control library that also manages access to a parameterised Bluespec IP library. We demonstrate the potential of our infrastructure with two use cases with unmodified application executables where, 1), an executable is dynamically instrumented to generate load/store and program counter events that are sent to FPGA hardware accelerated in-order microarchitecture pipeline, and memory hierarchy models, and 2), the design of a branch predictor is prototyped using an FPGA. The key features of our infrastructure are the ability to instrument at instruction level granularity, to code exclusively at the user level, and to dynamically discover and use available hardware models at run time, thus, we enable software developers to rapidly investigate and evaluate parameterised Bluespec microarchitecture and accelerator IP models. We present a comparison between our system and GEM5, the industry standard ARM architecture simulator, to demonstrate accuracy and relative performance; even though our system is implemented on an Xilinx Zynq 7000 FPGA board with tightly coupled FPGA and ARM Cortex A9 processors, it outperforms GEM5 running on a Xeon with 32GBs of RAM (400x vs 700x slowdown over native execution).**

I. Introduction

Computer architects rely extensively on simulators, in order to evaluate, and identify, the key software and hardware aspects, affecting the performance, and power consumption of future architectures, on current, and emergent application use-cases. *System call emulation* mode simulation of user-level application code is sufficiently accurate for many application use cases, such as the SPEC CPU benchmarks [1], where user-level application code execution dominates behaviour, and OS kernel level code can largely be ignored. The need to be able to study large realistic application use-cases within reasonable time-frames has led to the use of reduced accuracy [2], or analytical [3] models of microarchitecture and memory systems hierarchy in order to reduce computational complexity. Notable simulation systems include [4], [5], [6], [2], [7], [8], [9].

Computer architects need to incorporate accelerator IP into their simulation system, preferably using the synthesised IP hardware itself, as interfacing a functional model of IP to a traditional software based simulator such as GEM5 [9] increases the possibility that design errors will not be caught early in the development cycle. Unfortunately, the software simulation of synthesisable *register transfer level* (RTL) is very slow, and initial testing of accelerator IP using RTL simulation is often limited to traces captured from real application executions that may not be representative of all input data sets. Further, compiler and toolchain developers require facilities for fast simulation and testing of accelerator IP that modifies the *instruction set architecture* (ISA) and/or, the internal microarchitecture, because their toolchains must generate optimal code sequences to exploit new instructions and internal microarchitecture functionality.

In this paper we present a methodology to combine the accuracy of synthesised IP hardware models with the flexibility to simulate arbitrary binaries using software based simulators. Here, we fully decouple functional simulation from the timing models used to produce performance information. Dynamic binary instrumentation of an unmodified application executable is used to generate an event stream comprised of the addresses of data loaded and stored, along with the program control flow addresses taken by native execution of user-level threads. The event stream is consumed by timing models that are implemented in Bluespec and synthesised to an FPGA. Our current system targets in-order ARMv7 ISAs using a Xilinx Zynq board with dual-core Cortex A9 processor cores and a tightly coupled Xilinx FPGA. Our system could easily be retargeted to X86 ISAs by using an additional level of dynamic binary translation, or by directly exploiting Intel's Pin tool for dynamic binary translation of the X86 ISA using the Intel Xeon Accelerator Platform, where X86 cores are tightly coupled with an Altera FPGA. In our system, accelerator IP can be directly plugged into the simulation system in the form of a synthesisable model encapsulated in Bluespec, thus we are able to directly test the overall system performance on an unmodified application executable. Note, if the IP concerns support for a modified ISA with new instructions, then the application executable must be compiled to exploit the modified ISA, and a dynamic binary modification plugin needs to be developed to interface the new instructions to the Bluespec encapsulated IP.

In this paper we detail the following contributions:

- We outline the design, flexible modelling capabilities, and performance of our dynamic binary instrumentation based simulation system that is constructed using a Bluespec library of composable timing models and the open source MAMBO [10] dynamic binary instrumentation system.

We present relative performance information that compares our system to GEM5's system call emulation mode.
- We demonstrate, using a prototype branch predictor example, how our infrastructure enables accelerator IP to be directly integrated with our simulation system, in order to evaluate its performance on real application executions without the need to generate traces.

In section II we review related simulation techniques. In section III we introduce the design and methodology of the infrastructure. Sections IV and V outline use cases and implementation issues respectively. Section VI presents the experimental method and an evaluation of the results produced. Section VII discusses conclusion and future work.

II. Related Work

In this section we discuss the approaches of GEM5, Sniper, ZSim that are software based, and FAST and HAsim that utilise FPGA based, or assisted simulation systems for microarchitecture. An in-depth survey of FPGA accelerated simulation of computer systems is contained in [11].

A. Software only simulation

GEM5 [9] is currently the de-facto standard for architecture simulations, and the only open source simulator for ARM based systems. GEM5 provides a highly configurable simulation infrastructure handling multiple ISAs, and CPU models (ranging from functional-only atomic models to cycle-detailed *out-of-order* (OOO) models) in conjunction with a detailed and flexible memory hierarchy providing support for multiple cache coherency protocols and interconnect models. The simulation mode can be either *system call emulation (SE)* mode, concentrating on the simulation of user-level application code, or, *full-system (FS)* mode, where hardware devices are modelled and unmodified operating systems are booted, here, both user and kernel-level instructions are simulated. SE mode is suitable for workloads and benchmarks where user-level computation dominates. FS mode is suitable for cases where OS services, or access to I/O devices have a significant impact on overall application performance.

Sniper & ZSim [6], [12], [3] both utilise Intel's Pin infrastructure to dynamically rewrite the code executed by X86 applications using a *pintool*. Sniper and ZSim use custom pintools to implement their simulation infrastructures by extracting key information from an appplication's execution. **ZSim** decodes instructions to μ-ops related to the pipelined microarchitecture under simulation. A core's pipeline stages are simulated and evaluated at each μ-op. **Sniper** uses interval simulation based on high-abstraction analytical models of core performance. A core's instruction stream is divided into timing intervals delineated by miss events, such as branch mispredictions, and cache misses. An interval's performance is determined by analysing all of its instructions, and the penalty of its terminating miss event. In [3] a sampling based methodology is used to reduce the amount of time spent simulating an application in detail to less than 10% of the total application runtime by identifying the periods when faster less accurate simulation models may be applied without significantly affecting accuracy.

B. FPGA-based simulation

FAST's [13], [14] key contribution is that it was one of the first systems to use FPGA accelerated timing models implemented in Bluespec, that are driven by speculative functional execution of full-system simulation using modified QEMU software. QEMU determines the dynamic instruction trace passed to the timing models. However, at program execution points where a branch mis-speculation occurs, functionally incorrect instructions will be fetched in a real processor until a mis-speculation is resolved, then the functionally correct branch path is followed. Therefore, the modified QEMU software must use costly check-pointing at places where such mis-speculations can occur, and roll-backs to a checkpointed state when a mis-speculation occurs in order to follow mis-speculated instruction traces until they are resolved. Such modifications further slow down the execution of a vanilla QEMU that is already significantly slower than other dynamic binary instrumentation tools such as MAMBO and Pin. It is important to note that FS simulation is unnecessary for many applications, and that the infrastructure described in this paper enables significantly more flexible interfacing to timing models than FAST, as well as the integration of accelerator IP.

HAsim [8] is a novel FPGA-based simulator that simulates multicore architectures using a highly-detailed processor pipeline, cache hierarchy and detailed on-chip network using a single FPGA. Its main contribution is the use of fine-grain multiplexing of pipeline models to support detailed timing for multicore processors. Internal core state such as the PC and register file is duplicated but the combinational logic used to simulate each pipeline stage is not. HAsim's timing models track how many FPGA clock cycles represent a single target machine cycle for a given operation. In this scenario, structures such as cache memories that do not map efficiently to FPGA resources, are implemented using FPGA efficient features (such as Block-RAMs), and multiple FPGA clock cycles then represent a single target machine cycle. A port based connection model is used to resolve the total target machine cycles when interactions between different microarchitectural timing models are necessary.

III. Infrastructure Design & Methodology

In this paper we present the APTSim simulator for ARM architectures. APTSim is able to simulate arbitrary ARMv7 binaries using hardware models for memory system hierarchies and microarchitecture pipelines. It is built from two main components: our novel simulation infrastructure MAST, and the dynamic binary modification tool MAMBO. Figure 1 overviews the main components of the simulation infrastructure, and how it generates performance estimates and statistics from a simulation. MAMBO [10] is an open source dynamic binary modification tool for ARMv7, that we extended with custom plugins to implement a functional

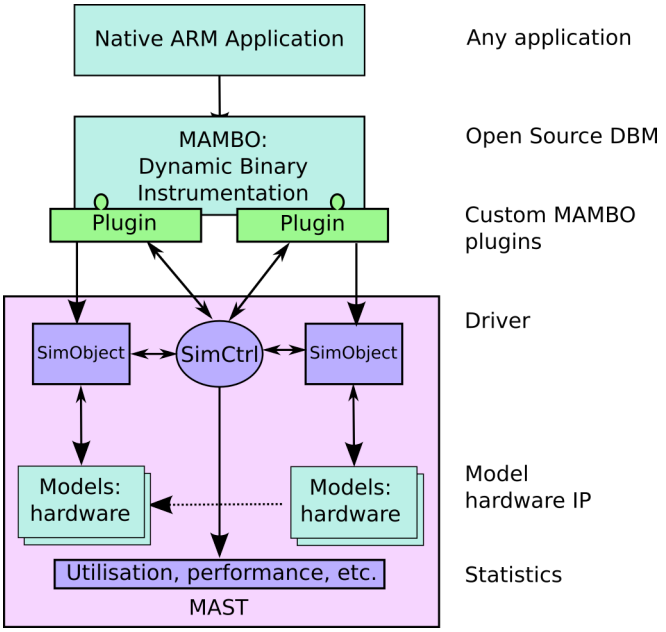

Fig. 1. APTSim Overview.

simulation of an ARM application execution; our custom plugins generate streams of events (memory accesses and PC-altering instructions), that feed MAST memory system and pipeline models. MAST enables Bluespec IP hardware models to be managed by a flexible userspace software driver library written in C++. The advantage of this approach is that it is feasible to deliver cycle level accurate timing at high performance, because the most significant cause of slowdown is limited by dynamic instrumentation overheads rather than the timing emulation overheads on the FPGA that are minimal in comparison. Moreover, by keeping the functional model in software, decoupled from the hardware IP, the infrastructure is flexible and easy to extend. In the rest of this section, we describe the IP library, the software userspace library, and MAMBO-based instrumentation.

A. MAST BLUESPEC IP LIBRARY

MAST uses Bluespec to facilitate the rapid construction of highly parametrised models with well defined interfaces. Currently the library consists of a number of high level models and a selection of lower level IP blocks that aid model implementation, for example ARM's AXI IP, that enables straightforward creation of IP blocks with Xilinx Vivado, using AXI as the default network protocol. Most MAST models have a single AXI interface, controlled by a library IP block that handles its identification, and locking. Identification currently uses a 32-bit word consisting of 20 bits for a model "type", 4 bits to specify a version and 8 bits to identify model specific features; which can be used to control how the derivative models can accessed by a common software driver. Bluespec's atomic rule based coding style enables the synthesis of control systems that would be complex to specify using RTL languages. A MAST compliant IP component must adhere to a low-level IP interface containing identification, locking, data movement and IO features such as burst controllers for fetching data from processor memory, and file based reading/writing IP. The high level models are currently: a *Basic Cache Model*, *Snoop Controller*, *Cache Systems*, and a *Pipeline Model*.

a) Basic Cache Model: gathers statistics about the behaviour of a cache system. It is not a functional model, i.e. it does not store data, only address tags and states for cache lines, this allows the behaviour of a cache to be evaluated whilst the area of the model remains manageable. The model always features an AXI slave that is used to lock the model to an application (the simulator), and to read back statistics; additionally, an optional ACP interface may be also included, for burst transfers from the CPU, snooping and requesting data from other caches. The cache model parameters specified at generation time determine features such as cache size, associativity and block-size. Each cache is constructed of a number of *way* models, each using a separate block RAM. When an address is sent to the model, from any port, each way is accessed in parallel to see if the address is present, if it is, a hit is counted and the relevant row in each way is updated to keep track of the order in which they have been accessed. Should a miss occur, a free way will be used if possible, or one of the ways will be evicted, based on the replacement algorithm (the least recently used or a random selection), configured at run time. Misses and evictions are typically referred out from the cache model to a snoop controller, or a higher level cache. Statistics are maintained for more detailed analysis. To gather statistics for an L1 cache, we need to provide the model with a sequence of addresses, each corresponding to a memory access, along with the type of access. For higher level caches all information flows from the L1 caches, so the CPU does not need to provide any input, although statistics are read directly from that model. The model also includes a TLB per cache, a feature of ARM A7 in-order cores (and other ARM designs). The current version of the cache models a write-back policy.

b) Snoop Controller (SCU): takes input requests from a number of cache models, and ensures coherency is maintained between them, currently using the MESI protocol. This involves snooping other L1 caches, and updating their coherence state where required. The model passes transactions up to higher level caches if the data required is not present in any L1 cache. The model determines if the hierarchy is inclusive or exclusive. An SCU is not owned by a specific application, it accepts data from any attached caches, and does not maintain state or counters, hence it does not require interaction from the CPU and has no AXI port. Statistics related to the coherency protocol are maintained in the L1 caches indicating how often the cache has been snooped, and if the snoop has impacted on its state.

c) Cache Systems: facilitate the simulation of large memory hierarchies. There is, currently, a hardware restriction of a maximum of four master models per system, a cache system model can contain many caches but is a single master model. Two common cache structures are included as additional

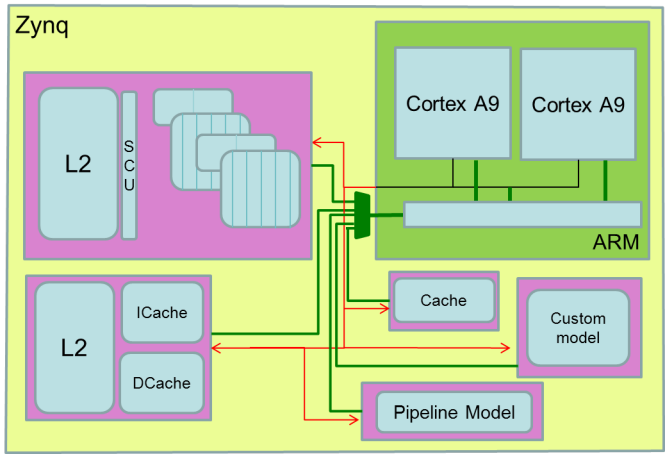

Fig. 2. Hardware system modelling.

models in the library. The first is a dual L1 cache, with a shared L2 cache, common in Intel-type multicore systems. The second is a cluster cache hierarchy, with a parameterised number of core caches (L1 instruction + L1 data) coupled via an SCU to an L2 cache, this is the structure found in ARM's big.LITTLE style architecture. The advantage of these combined models is that their components share a common ACP port, allowing larger memory systems to be configured on a device, they do however have a significant disadvantage in that as our methodology involves creating IP blocks it is not possible to probe within running models and this makes IP block debugging more challenging.

d) Pipeline Model: consists of an ARMv7 instruction decoder, and a model of the pipeline for an in-order ARM Cortex-A7 dual-issue processor, with five functional units. Additionally it contains a number of block RAMs which are used to cache pages of program memory. The model does not functionally model the pipeline, i.e. it is not a functioning processor, it simply decodes instructions, counting individual instruction types, and registers used, and allocates instructions to a processor pipeline to obtain timing and utilisation statistics. The model avoids hazards by tracking register reads and writes, and the progress of the instructions on functional unit pipelines. The in-order issue logic stalls if necessary. The model currently streams instructions from the block RAMs, being given a start address by the host CPU, and continuing evaluating instructions until a branch is hit, or the next address is not stored in the model (a local page fault), at which point it waits for the CPU to provide the jump address, or a new page of instruction data. The pipeline model can communicate with a cache model to provide requests to the instruction cache. To gather statistics from the executing program we need to provide the model with addresses, typically this is done from a dynamically instrumented executable, although we may provide them from a trace file. This functionality enables the debugging and validation of our infrastructure against traces from other simulators such as GEM5. The model currently counts over 350 ARM and Thumb mode instructions types, all register accesses and various other microarchitectural features, each with a 64-bit counter; these counters are implemented using the LUT's shift registers for higher order bits, and counters for lower order bits, such that each LUT stores a single bit for up to 32 counters giving a relatively compact design but allowing for all counters to be updated at any time. These counters are mainly used to validate the accuracy of the simulator, and they can be removed to build smaller faster models.

Figure 2 shows an example system with instances of both cache systems (cluster cache hierarchy, on the top left, and Intel-like on the bottom left). The system also includes a pipeline model together with an isolated cache. Finally, this example also illustrates how MAST can be extended with new, custom models, as we describe in Section IV.

The flexibility of our Bluespec library is enhanced by a set of scripts that import Bluespec compiler produced Verilog directly into Vivado IP-XACT IP blocks; these blocks can be easily combined with a simple command to produce systems which monitor various architectural features of the modelled CPU. For the purpose of debugging we can automatically set up probing of AXI buses in the network.

B. MAST USERSPACE DRIVER LIBRARY

The hardware is managed by a C++ software library which acts as an entirely userspace driver for any IP blocks configured onto the FPGA. The library is able to recognise any IP blocks that are present, and to enable the appropriate plugins in MAMBO, allowing that correct instrumentation to be performed during program execution. It consists of two main classes which are used within any application accessing the FPGA, as shown in Figure 1. *SimCtrl* is responsible for managing the system as a whole and *SimObject* for controlling an IP block. The *SimCtrl* has a number of features allowing easy system development. On creation the object probes the FPGA to ascertain what hardware is configured on it; it creates a *SimObject* derived object for each IP block, which allows that IP block to be used by the application. The application can request access to a specific IP from the *SimCtrl* which will return, if available, an appropriate *SimObject*; the IP is deemed to be available if it is not locked or it is locked by a process/thread which is no longer running. The application is then able to gain ownership of IP by using the *SimObject* to lock the IP block to either its process or specific thread; this lock is stored in the hardware. Other applications, or threads, are unable to update the state of any IP blocks that they do not own, except to lock a free block.

SimObject is a base class used to access registers within an IP block. We derive from this class to form a specific IP abstraction layer between the hardware and the application developer; the base class allows only discovery, locking and low level access to registers on the IP; typically the *driver* developer for an IP block will add helper functions to wrap sequences of accesses to provide more accessible functions to the specific features of the IP block targeted. *SimObjects* can lookup physical addresses from virtual addresses using the

SimCtrl object, this allows IP blocks featuring AXI master ports to access processor memory directly, as FPGA IP can only directly use physical addresses. To send a data buffer to an IP block we can either send data a word at a time from the host ARM processor, or for increased performance, we send the physical address and size of a block to the IP, and allow it to directly access the processor memory, in a cache coherent fashion, via the ACP port. The use of the hardware IP master allows for dramatic performance increases over CPU managed transfer, whilst the ability to use an arbitrary block of memory makes the migration of CPU based code to FPGA accelerators a straightforward task, without having to deal with kernel level drivers.

C. MAMBO Instrumentation

MAMBO [10] is an efficient dynamic binary modification tool for ARM architectures that transparently modifies the machine code of 32 bit and 16 bit instructions during execution. Its performance is 2.8 times faster on average than Valgrind, and 14.9 times faster than QEMU. MAMBO is designed for behavioural transparency, meaning that it only implements the types and degrees of transparency required to correctly execute typical workloads that follow the platform ABI, use standard system libraries and do not depend on undefined behaviour as described in [15]. It is currently capable of running a wide range of unmodified applications, including the SPEC CPU2000/CPU2006, PARSEC multithreaded benchmark suite as well as large applications such as LibreOffice 4.2 and GIMP 2.8.

MAMBO operates by modifying program code at run time. Under MAMBO, applications execute from a software code cache populated at runtime/on-demand with the modified code. If execution starts at a point not present in the code cache then a basic block, i.e. a single-entry single-exit contiguous sequence of instructions, is translated from the original loaded program into the code cache; the next address will be a new basic block and the process repeats. When MAMBO populates the code cache it can modify the code in an arbitrary way thus allowing additional functionality to code execution.

To drive our hardware models we use MAMBO plugins, a flexible interface to add new functionality; these consist of a set of callbacks which are executed at various points of program execution. An initialisation function is used to assign end user provided functions to MAMBO events, in the case of APTSim we also use it to call *SimCtrl* to ensure that we have any hardware required by the plugin and thus enable plugin events. A pre-instruction callback is called before an instruction is inserted into the code cache, typically we would use this to insert a call to a function used to control hardware; for example, if the instruction is the first in a basic block we would call a function which sends the pipeline driver a `jumpAddress(*ptr)` to the source address, in the original code, for that instruction. Thus, every time a branch is seen in the functional model, the PC of the successor (either the target or the fall-through path) is pushed to the timing model. We also capture all memory accesses from load/store instructions. If the simulator is used only to study a single subsystem in isolation, a single stream of load-store accesses or branches can be generated. Else, both streams are created by the instrumentation tool. We might also remap instructions, for example if we try to execute a GEM5 special instruction, which would ordinarily cause a native application to terminate with an illegal instruction; in this case we have inserted an instruction handler to make APTSim execute an equivalent function, e.g. reset or dump statistics, and remove the "illegal" instruction from the executing code. Instruction remapping can be used to support ISA modifications, where MAMBO plugins, and the userspace driver cooperate with accelerator/new instruction IP added to the Bluespec library in order to enable seamless execution (to the user) of new instructions and accelerator functional units.

IV. USE-CASES

We present two additional use cases that demonstrate the inherent flexibility of the infrastructure to undertake more use-cases than just simulation by leveraging MAST.

A. MICROARCHITECTURAL PROTOTYPING

It is often unnecessary to implement (or use) a complete simulator for the purpose of prototyping microarchitecture, as it is only necessary to support the specific features that are not currently provided. For example, a researcher exploring design options for a branch predictor need only model the predictor's performance, or it may be of interest to prototype an altered ISA supporting new instructions. As mentioned earlier, instruction remapping using MAST with MAMBO plugins can enable the new/modified functionality ISA instructions to be seamlessly executed using new hardware, modelled in the FPGA, whilst the remaining code runs natively.

The single major advantage is the ability to use arbitrary binaries to test and evaluate prototype hardware, rather than using testbeds created from traces. Further, the reduced area requirements in comparison to a complete simulator, make it feasible to model n multiple alternative designs of the microarchitecture in parallel during a single application execution. This is a significant advantage for design space exploration, effectively dividing by n the time required, or multiplying by n the number of design points that are explored. We illustrate this use case in Section VI-C where we model four different implementations of a branch predictor unit in parallel.

B. ACCELERATOR PROTOTYPING

MAST hardware and software libraries are not exclusively applicable to binary instrumented designs. It is easy to implement more conventional accelerators, such as image processing filters, and access these from regular applications. For example, we have implemented the front-end filter stages of a KFusion SLAM application [16] using Bluespec. Images are passed to and from the main applications using the *SimCtrl* unit to map pages from virtual to physical addresses and storing page mappings on the FPGA, as a translation lookaside buffer (TLB). This use-case demonstrates that only user-space

coding is required. Furthermore the application can check for the presence of the accelerator at run time and use it if available; in principle the application might reconfigure the FPGA, assuming that the *SimCtrl* reported no other IP blocks were in use.

V. IMPLEMENTATION AND FRAMEWORK DESIGN ISSUES

APTSim uses a Xilinx ZC706 evaluation board running Ubuntu 14.04 with 1GB DRAM (no swap), a Zynq-7000 XC7Z045, and dual 667MHz ARM Cortex-A9 processors. Reduced size datasets are used for some benchmarks, as we must fit our simulation runtime infrastructure within the 1GB RAM space.

For an end user of APTSim the process of running a simulation is slightly more complicated than for a conventional simulator as they must also produce a hardware configuration to match their desired memory hierarchy and pipeline models. Assuming that the model has not been created before, the user must first make specialised models for their system; since typically the generic component exists in the MAST library, this is a relatively straight forward process, for example the code below will create a 2 core cluster with the caches: L1 32K 4-way with an 8-entry 1-way TLB and L2 512K 8-way with a 32-entry 4-way TLB.

```
import mast::*
(* synthesize *)
module myCluster(AxiACP\_ifc);
  AxiACPbigLITTLEClusterPoly_ifc#(
   2,                //2cores
   32768, 4, 8, 1, //L1 Instruction Cache
   32768, 4, 8, 1, //L1 Data Cache
   52488, 8, 32, 4,//L2 Cache
   32, 32,           //32 bit data/address
   32                //32 byte cache lines
  ) c <- bigLITTLEClusterModel();
  interface sread = c.sread;
  interface swrite = c.swrite;
  interface mread = c.mread;
  interface mwrite = c.mwrite;
endmodule
```

Running the source through the Bluespec compiler will produce a Verilog file which is suitable for creating a Vivado IP block. To produce a system containing the cluster IP and two pipeline models the following can be loaded into Vivado; additional models can be instantiated by adding them to the model list.

```
source ${MAST_HOME}/../tools/vivado/mast.tcl
mast::package_ip myCluster.v
mast::test_ip {myCluster pipelineModel
    pipelineModel} 75 simple
```

For the purposes of this evaluation we constructed the system described above. Table I shows the utilisation reported by Vivado after the implementation step.

The current implementation used in this evaluation is focussed on validation rather than on performance or area. For example, the overall system of IP models currently run at a relatively slow 75MHz. The critical timing path is limited by counters for validation within the pipeline models. Currently, we record statistics on approximately 350 different instructions, and the counters occupy a significant area. In most simulation and modelling use-cases, most of the counters could be removed, as total accumulated cycle time is more relevant. However, the timers are essential to validate that the simulator is correctly processing the executable. Further, addresses are currently inefficiently transferred one by one to the memory system models. This causes slow transfers that block the processor for several cycles. The memory models have the capability to act as masters, this allows for transactions to be logged using double buffers, and DMA-ed from one buffer whilst another is concurrently filled, with the additional benefit of faster data transfers, and allowing the host CPU to continue generating trace data whilst the models process the existing data.

Component	LUT/model	BRAM/model
L1 I+D cache models	$6110*2$	$8*2$
SCU	327	0
L2 Cache model	3228	16
Pipeline models	$14904*2$	$4*2$
Network	5179	0
Utilisation	21.87%	5.13%

TABLE I
FPGA UTILISATION FOR THE EVALUATION SYSTEM.

VI. EXPERIMENTAL EVALUATION

To evaluate the accuracy of the models we have two options: we can compare against an existing simulator or we can compare against performance counters on an existing processor. The latter is challenging on ARM due to the restricted availability of such counters on most ARM SoCs. Thus, in our system we use GEM5, the *de facto* standard simulator for ARM, in system call emulation mode to generate statistics and compare against APTSim; in particular we are using the *atomicSimple* CPU model and, this combination provides maximum performance. We are currently only evaluating single threaded applications, whilst APTSim is able to handle multiple threads, and the system implemented is multicore, the complexity added to validation is excessive at this stage of development. Whilst the use of GEM5 in syscall mode requires the use of static binaries this is not a prerequisite for APTSim.

A. Validation

The initial phase of validation was focussed on resolving issues with MAST hardware/software libraries, and MAMBO plugin functionality, consequently we used benchmarks from the Problem-Based Benchmark Suite [17] to provide small targeted tests with relatively simple assembler structure. Later, we have tested the accuracy of the infrastructure with larger more complex programs from the standard SPEC CPU2006 benchmark suite [1].

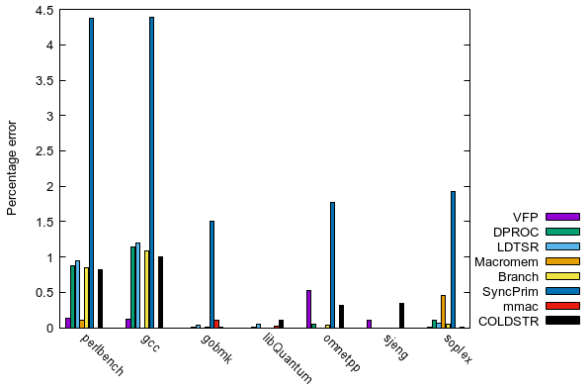

Fig. 3. Comparison of counter class between GEM5 and APTSim for test SPEC 2006 benchmarks.

Although the functional execution on APTSim matches execution on GEM5, we have observed minor divergences in floating point code, and in time-dependent behaviour (e.g. loops around load/store exclusive instructions). To minimise unknowns from the comparison, we ran GEM5 with a PC tracker to produce a trace address after any potentially PC changing event, that is to say jumps, branches and instructions with PC as the target. This behaviour is the same as we would expect from MAMBO except we have a guarantee that the execution path is exactly the same as for GEM5. We then load the executable, using an ELF loader, and pass the address events stream to the MAST library, whilst this does not conduct a functional simulation it does provide the ability to gather statistics on instruction mix, register usage, etc. Any unexpected instruction encodings are easily detected, as the instruction addresses are flagged to the MAST library and recorded as an error. Using this method it was possible to validate the instruction decoder aspects of the pipeline model. Experiments were conducted using both ARM and Thumb compiled code to ensure a wide instruction coverage using a range of SPEC2006 benchmarks, repeated until there was an exact match between GEM5 and APTSim statistics. This methodology shows the flexibility of MAST and the potential to use it as an accelerator for other simulators.

On completion of the trace validation we returned to executing various SPEC2006 benchmarks reassured that the deviation in instruction counts is a function of varying instruction paths rather than errors in the models. We ran each benchmark with test data, rather than full data sets, to maintain reasonable simulation times and as the simulation platform has insufficient memory for some of the tests. Figure 3 shows the accuracy of the results, calculated as the sum of absolute error for each instruction, grouped by category. Whilst the error is typically less than 1% there is a tendency for APTSim to have higher counts than GEM5, we believe this is a result of the "ideal" nature of the GEM5 configuration versus the more realistic behaviour of APTSim, hence the larger discrepancy in synchronisation instructions. We consider the discrepancy between the results to be reasonable.

B. Performance

Our current work has been on accuracy of simulation, rather than performance and there is significant scope for performance improvement. Current simulation performance is shown in Table II. For APTSim we have included performance for the combined pipeline and cache system, but also for a system only containing a memory system, this allows evaluation of the memory hierarchy with a reduced performance penalty. Results for GEM5 were obtained with one of the simplest and fastest models provided, on an Intel Xeon E3 3.2GHz with 32GB of RAM.

There are significant opportunities for performance improvement. Firstly the use of page sized RAM buffers in the timing model means that if code is not present in a RAM buffer then an entire page is written to the FPGA RAM; in many applications more than 4 pages of RAM will be used in relatively tight blocks of code meaning time is wasted on moving data to the FPGA. A logical alternative solution is to directly access the specific memory over the ACP port, using the processors cache as a code cache, this could significantly reduce the amount of communications to main memory and cause less pollution of the processors cache, both of which should improve overall performance. Moreover, the memory model is currently transferring accesses an address at a time to the cache model, a more rapid solution would be to double buffer these in the MAMBO plugin and allow the cache to DMA them from the buffers as they become full; this would allow overlap of cache modelling and data gathering and the use of ACP offers around an order of magnitude in data transfer rates compared with direct write.

Simulation method	Slowdown factor
Native, on Zynq board	1
APTSim Cache+pipeline	400
APTSim Cache	250
GEM5 syscall atomic	700

TABLE II
RELATIVE SLOWDOWN FOR GEM5 AND APTSIM.

C. Microarchitecture Prototyping

We implement and evaluate a branch predictor unit to illustrate the potential of the infrastructure to prototype microarchitectural features and perform design space exploration. Branch predictors are used in speculative execution processors to predict the path taken by PC-altering instructions. Modern microarchitectures, with long pipelines and out-of-order execution, may require a high number of cycles to resolve the actual target of a branch instruction that should be executed next. Once the target PC successor to a branch has been speculated, then the instruction fetch unit can continue to operate from the predicted PC, that is eventually compared with the actual outcome, and on a misprediction, the fetched instructions are discarded.

We study how the size of the prediction tables affect the performance of a simple Bimodal branch predictor, and for Tournament, a more complex branch predictor. Figure 4 shows

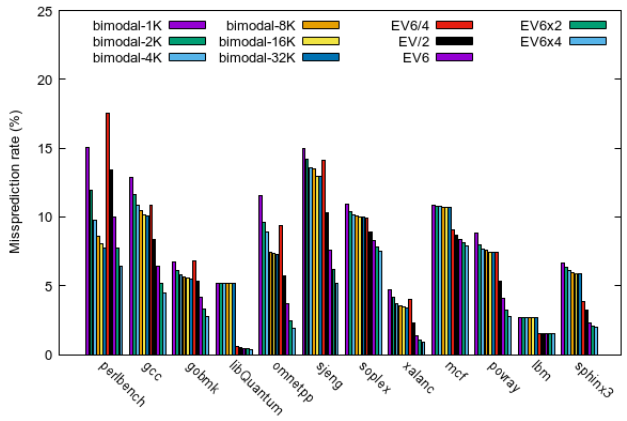

Fig. 4. Misprediction rate of various branch predictors for test SPEC 2006 benchmarks.

the misprediction rate of various Bimodal and Tournament branch predictor sizes. We present results for a subset of the SPEC CPU 2006 benchmarks fitting within the available memory footprint of the Zynq platform. To speed up design space exploration, we grouped multiple predictors into a single model. We tested all the predictors of one model simultaneously in a single run. Bimodal-*n* has an *n*-entry table of 2-bit counters. EV6 replicates the configuration of the branch predictor in the Alpha 21264 processor, with a 1024-entry local history table, a 1024-entry local predictor with 3-bit counters, a 4096-entry global predictor with 2-bit counters, and a 4096-entry choice predictor with 2-bit counters, for a total of 29Kbits. Other Tournament design points (see the figure key) scale down or up the number of entries in the tables accordingly.

We observe a wide range of behaviour for the different benchmarks, with miss rates between 17.5% and 0.4%. Some benchmarks benefit from larger tables, but not all of them. Most benchmarks benefit from the ability of Tournament to take into account history, although `perl` shows lower misprediction rates for bimodal if using similar area budgets. `perl` runs in a relatively short time, so it is plausible that this happens because of the longer time required to warm up Tournament's more complex structures in comparison to bimodal. Even such a limited exploration of the design space shows how branch predictor units can have complex behaviour. In future work, we plan to implement and study a wider collection of designs (e.g. Perceptron or TAGE).

VII. Conclusions & Future Work

We have demonstrated the potential of combining a flexible IP hardware library, a user-level driver library and dynamic binary instrumentation for microarchitecture simulation and prototyping. We exploit the advantages of FPGA SoC to accelerate at a very fine granularity (instructions), rather than large blocks of code. With this, we can benefit from the accuracy and speed of FPGA-based modelling and the ability to run arbitrary binaries. Moreover, this paper contributes the first FPGA-based simulator for ARM, significantly extending the options for simulating ARM processors.

As future work, we will implement the system on an Ultrascale Zynq board. Its features (e.g. faster CPUs, improved memory interface) make it an excellent platform to evaluate the performance of the proposal on a modern FPGA. Moreover, we plan to extend the models to include more microarchitectural features and alternatives (e.g. an out-of-order pipeline). Finally, we will connect MAST with a modified Java runtime that generates streams of events for the hot paths of the running binary.

Acknowledgements

This work was supported by EPSRC grants PAMELA EP/K008730/1 and DOME EP/J016330/1. Mikel Luján is funded by a Royal Society University Research Fellowship. Oscar Palomar is funded by a Royal Society Newton International Fellowship.

References

[1] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.
[2] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA*, 2013.
[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *ISPASS*, 2013, pp. 2–12.
[4] E. Argollo *et al.*, "Cotson: Infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, Jan. 2009.
[5] Z. Tan *et al.*, "A case for fame: Fpga architecture model execution," in *ISCA*, 2010, pp. 290–301.
[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*, 2011, pp. 52:1–52:12.
[7] Z. Tan *et al.*, "DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 207–221, Mar. 2015.
[8] M. Pellauer *et al.*, "HAsim: FPGA-based High-detail Multicore Simulation Using Time-division Multiplexing," in *HPCA*, 2011, pp. 406–417.
[9] N. Binkert *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[10] C. Gorgovan, A. d'Antras, and M. Luján, "MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM," *ACM TACO*, vol. 13, no. 1, pp. 14:1–14:26, Apr. 2016.
[11] H. Angepat *et al.*, *FPGA-Accelerated Simulation of Computer Systems*, ser. Synthesis Lectures on Computer Architecture, 2014, vol. 9, no. 2.
[12] W. Heirman *et al.*, "Power-aware Multi-core Simulation for Early Design Stage Hardware/Software Co-optimization," in *PACT*, 2012, pp. 3–12.
[13] D. Chiou *et al.*, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *MICRO*, 2007, pp. 249–261.
[14] D. Chiou *et al.*, "The FAST Methodology for High-speed SoC/Computer Simulation," in *ICCAD*, 2007, pp. 295–302.
[15] ARM, "ARM Architecture Reference Manual, ARMv8, for ARMv8-A Architecture Profile." 2015. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.i/index.html.
[16] R. A. Newcombe *et al.*, "KinectFusion: Real-time dense surface mapping and tracking," in *ISMAR*, 2011, pp. 127–136.
[17] J. Shun *et al.*, "Brief announcement: the problem based benchmark suite," in *SPAA*, 2012, pp. 68–70.