HLS^2 : High-Level Synthesis for High-Level Simulation using FPGAs

Varun Koyyalagunta, Hari Angepat and Derek Chiou
University of Texas at Austin
{varun,angepat,derek}@fast.ece.utexas.edu

Abstract

With the growing capacities of modern FPGA fabrics, the use of FPGA-based computer system simulators has become an increasingly mainstream option to increase scalability and throughput. Current state-of-the-art simulation platforms typically require manual writing of custom FPGA-logic to implement the required simulation tasks leading to extensive development time. We evaluate the use of high-level synthesis tools (HLS) as applied to existing high-level simulation models as a means to reduce this development time. We find that after restructuring the model for FPGA-based computation, a straightforward C-to-gate HLS toolflow can still provide reasonable throughput compared to a pure software simulation model. Furthermore, such a strategy provides a means of quickly moving to a FPGA-based simulation platform, allowing researchers to concentrate their development effort on only the modules of interest while not entirely sacrificing fidelity on modules of less interest.

General Terms Interval Modeling, High-Level Simulation, High-Level Synthesis

Keywords FPGA, Interval Models, HLS

1. Introduction

With the continuing push towards increasing levels of parallelism in next generation computer systems, the need to quickly evaluate such systems has become a pressing need for researchers. To support both the scale and speed requirements of such simulation models, a number of research groups have turned towards FPGAs as a simulation platform [5, 1, 6, 2, 6, 8, 9]. These FPGA simulators are typically clean-slate implementations that heavily leverage FPGA-based optimizations to optimize for the resource constraints imposed by the simulation fabric. While many order of magnitude speedups can be achieved using such techniques, they currently require explicit knowledge of targeting an FPGA for reasonable results.

In this work we explore high-level synthesis as an automatic means of transforming software-based simulators into FPGA-based simulator modules. As transforming an entire simulator code base that makes extensive use of high level software primitives (vectors, hashtables, linked-lists, malloc/free, etc) would require significant engineering effort with current tool maturity, we focus on a restricted case study. We explore the use of transforming a recently proposed high-level simulation model, interval models [3], for modeling out-of-order processor cores. Such a model can be used as a black-box substitution module for out-of-order cores when studying uncores, reducing FPGA occupancy for portions of the simulation that are not under detailed study.

1.1 FPGA-based Simulators

While FPGA capacity has continued to grow, so has the number of research groups building FPGA-based simulator targeting many-core processor/core/network models [5, 1, 6, 2, 7, 8, 9]. These simulator frameworks offer the ability to simulate at high-fidelity models incurring intractable simulation speeds. Implementing such a simulator in the same manner as a software simulator would result in a resource/timing explosion. As a result, such systems have typically been aggressively optimized for FPGAs to maximize throughput in terms of simulated cycles per FPGA resource. One such optimization commonly used by a number of projects is host-multithreading [2, 6, 7] that makes much more efficient use of hardware by mapping multiple target cores onto a single host pipeline. Such FPGA-specific optimizations can be costly in developer time, and as a result, there is still a significant productivity gap that exists when comparing traditional software simulators (typically implemented in C/C++) and simulators specifically written for FPGAs. This productivity gap has been addressed in some part by decoupling functionality from timing [1, 6, 7], using FPGAs for accelerated sampling [2], and aggressive modular component reuse [6]. However compared to contemporary sequential software development, the fundamentally additional constraints of FPGA area, routing, and timing closure add significant overhead to development time.

1.2 FPGA Abstract Models

Transforming an existing C/C++ detailed simulation model to an FPGA-optimized structure is a significant engineering effort. As a result many clean-slate FPGA simulator

frameworks only implement the specific models they require for a particular target under study, rather than porting a complete suite of existing models. For example, many detailed network-on-chip simulators implemented in FPGAs forgo the complexity of modeling a CMP directly and replace the entire core/cache for each node with a simple traffic generator[8]. This manages both the resource and development complexity of the FPGA simulator but comes at the cost of flexibility and accuracy. A high-level simulation model can offer a compromise between model complexity and accuracy. By not directly modeling the structure of a target, a high level model can capture the salient performance features of a module with reduced design complexity. In particular, for FPGA-based simulation, such models provide a middle ground between detailed models for the entire system and unreasonably simplistic models (e.g. core models with fixed CPI).

While it is possible to implement such models directly in an existing FPGA framework, this paper examines the opportunities available via High-Level Synthesis (HLS) in transforming existing software simulation models for use in FPGA-based simulation. Ideally if a HLS tool could generate FPGA implementations of high-level models for less critical target components, one can restrict FPGA specific optimizations and development time to the portions of the simulation that are actively under study. Further, by using a rapid prototyping HLS tool to implement the models, it becomes possible to iteratively customize the model to suit the particular details for a given study, generating the FPGA model automatically. By leveraging reduced complexity models for certain models we try to enable

- Spending crucial FPGA resources on parts of target under detailed study without dramatically impacting accuracy
- Automatic transformation of high-level models using high-level synthesis tools

2. Interval Modeling

We base our high-level model case study on recent work in high-level interval models by Genbrugge et al. [3]. Interval based modeling attempts to raise the abstraction level of OOO core modeling while still accurately simulating the intrinsic performance entanglement of co-executing threads via the memory subsystem. The basic technique is to provide a high-level mechanistic model for the processor core and a standard memory hierarchy model.

An interval-based processor model is based on the premise that OOO cores process instructions at a rate equal to the core issue width, disrupted only by long-latency miss events (L1 icache misses, L2 dcache misses, TLB misses, branch mispredicts, serializing instructions). The commit instruction stream is also rate limited by the intrinsic data dependencies of the instruction stream (critical path). An example of this basic interval timeline model is shown in Fig 1.

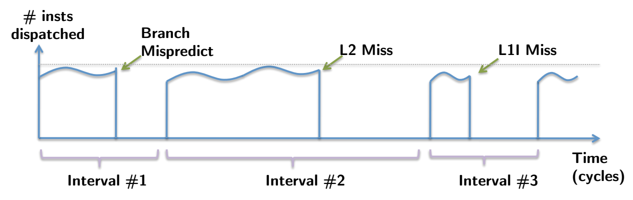

Figure 1. Interval Timeline [3]

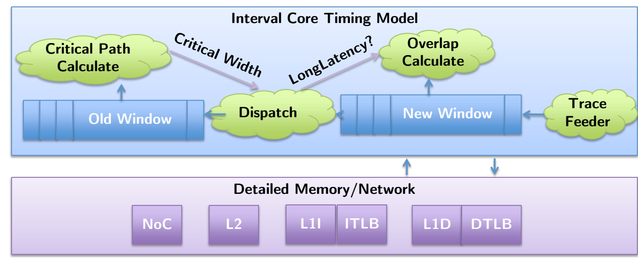

Figure 2. Interval Block Model

The model is trace-driven with the timing model split into a new window (which is used to model the ROB depth and supports overlap analysis), and an old window (which is used to model the critical path of the instruction stream). The remaining portions of the system can be modeled at whatever the required resolution for the experiment under study (see Fig. 2).

As a OOO core can support multiple misses outstanding and hide the effective miss penalty, such behavior is modeled via overlapping miss analysis. When a long latency miss hits the head of the new window, the contents of the new window are scanned. Any independent miss events found in the window are marked with as being ‘overlapped’ by a previously observed miss event. This allows handling the case where a L2 miss penalty is hidden by a previously issued L2 miss, preventing the naive serialization of penalties when the instruction hits the window head.

To support dynamic instruction dispatch width, the old window is used to compute the critical path of the previously ‘committed’ instructions. Each instruction inserted into the old window is timestamped as the earliest timestamp its inputs were available plus the actual execution latency of the op itself. By keeping track of the youngest instruction to enter and leave the old window, we can approximate the critical path length as the simple difference between these two. Finally Little’s law allows us to compute the dynamic dispatch width for a given cycle as the min(machine-width, old-window-size/critical-path-length).

The benefits of using such a model for FPGA-based computation are reduced complexity (both in development time and resources) while preserving a high-level of accuracy for the OOO core model. As the model can be evaluated completely at run-time, it does not require constant fine-grain

monitoring and retraining. In addition, by constantly calculating the critical path and stall overlap on a single instruction granularity, the model is suitable for coupling with fine-grained memory/network models.

3. FPGA-Friendly Optimizations for Software Simulators

As current high-level synthesis tools are not mature enough to process entire complex software simulators, the software code must be optimized so that it lends itself to be translated to hardware. We discuss here some generic optimization techniques to be considered when transforming a C/C++ simulator so that it structurally fits on an FPGA. Such transformations are designed to be friendly for HLS but not optimized for a specific tool.

3.1 Data-Type Specialization

A typical software simulator may have one very large class representing an instruction or dynamic op that it operates on throughout the pipeline. The pointer to this superinstruction would be passed through the pipeline stages, with each stage only operating on a small subset of all the fields in the superinstruction.

As a high-level synthesis tool would have difficulty statically identifying all the fields that should or should not be sent to each stage, it may not be able to remove unused fields, leading to suboptimal efficiency. Instead of deriving from a large, monolithic data structure for every stage's input, specific structures are made for each stage's input that consist of only the elements that the particular stage needs.

This optimization can be done without too much overhead even in a complex program. For each stage, a structure simply needs to be defined containing only the required fields. Then this structure is used in place of the original, overarching superinstruction structure.

Another common characteristic of the data types in software simulators is that 64-bit timestamps are kept around in many structures. In most cases though, the full 64-bits is superfluous. Usually the program is interested in the time deltas between different point of executions, and in such cases, the timestamp only has to be as wide as the maximum delta that can occur.

Having 64-bit timestamps everywhere consumes unnecessary resources on an FPGA, and so the data-types that are used should be re-factored to only be as wide as they really need to be. Since C/C++ simulators often make good use of typedef when defining the datatypes, changing the bit-widths of data types can be done fairly painlessly.

3.2 Black-Box Component Substitution

As our high-level model must be able to be a drop-in replacement for other hardware core models, it must be able to live in a concurrent simulated environment. A high-level model might want to know immediately whether an instruction was a cache miss or a branch mispredict. However, if the cache and branch predictor are to be operated as separate concurrent modules, then the model must deal with the asynchronous nature of the requests, waiting in both real-time and target-time for the replies from those separate subsystems.

This imposes a new restriction on the originally proposed interval model which used future knowledge about the latency of a memory operation in order to determine if a long-latency event occurred for a given cycle. This knowledge took the form of a synchronous function call which allows the cache subsystem to advance in target-time and compute the total downstream miss latency without having the core advancing a target cycle. We modify the model to use a request/reply style communication pattern whenever a request is made to an external component (Dcache, DTLB, etc), which allows integration with the rest of the timing simulator (see Section 4.2).

When writing the software for different modules, such as the core, branch predictor, cache, etc., the modules should be loosely coupled, so that the modules can be cleanly separated and run concurrently.

3.3 Bounded Memory

In software, extremely large structures can be constructed for the purposes of algorithmic speed-up. For example, to avoid an O(n) search, a complex hash table could be made for O(1) look-ups. As a typically software simulator is not usually concerned with virtual memory footprint, but rather the actual working set of its critical loops, even if the table size is large, it may still be performant if the working set is kept small.

For an FPGA as HLS tools typically do not support either dynamic memory or caching data structures, the full memory footprint of such structures must be paid in at compile-time, making such structures intractable. As a result, we restrict ourselves to relatively primitive fixed-sized arrays for the majority of the required data structures. Such a restriction is not as unwieldy as it may seem as simulator code can typically mirror the structures of the hardware being modeled.

3.4 Common Path Resource Prioritization

Whether a large data structure should be used to get speed up should be decided by the speed up's affect on overall throughput. Following Ahmdal's Law, we construct custom data structures only in support of the common case. For the interval model it follows that we should attempt to design for the case where there is no miss event pending that requires a overlap computation. When such an event occurs, simulator throughput will be reduced temporarily as the overlap computation requires several sequential steps. However, as such events are infrequent, we bias the FPGA resources towards the common case.

4. FPGA Interval Models

Our goal was to implement a cycle-approximate processor core simulation model that would balance complexity and accuracy and which could be synthesized from software to hardware in a simple, push-button procedure. We chose the interval model as the basis for the mode, as it can achieve high accuracy (within 15.5% for the SPEC benchmarks) and low complexity (written in less than a thousand lines of C/C++) as reported from previously published work using a 64-bit Alpha ISA and a M5 OOO model with a dispatch width of 4 and a 256 deep ROB as a reference [3].

We implemented a software reference model using the basic principles from [3] to guide the design of the FPGA-friendly optimizations. To validate initial model correctness, we integrated the software model into PTLsim, a detailed x86 cycle-accurate simulator, for comparison purposes. We were able to obtain error rates as low as 3%, with an average of 18.2% on a subset of 14 SPEC2000 benchmarks using 100M instruction samples comparing to the OOO K8/Core2-style detailed model. As the interval model only accounts for branch mispredict, cache misses, and critical path as first order events, lower order effects are ignored. For the PTLsim OOO model, this included a restriction on the number of inflight branches to 16, that accounted for 20-40% of front-end stalls which we removed to more closely match the unlimited resource model assumed by the interval model. As our results are comparable to the original published results, we direct the reader to [3] for a more detailed analysis of model accuracy concerns and focus on the aspects of the design relevant to FPGAs for the remainder of the paper.

4.1 Data-Type Specialization

We specialize the datatype for each of the 3 primary FSMs (TraceFeeder, Dispatch, and CriticalPath) of the interval model. Additionally, this datatype specialization propogates into the large buffer structures feeding to/from these FSMs (NewWindow, OldWindow). We typedef all fields making up the datatypes to enable precise bit-width while enabling efficient software execution by simply modifying the typedef to an int.

For example, we define `RegIdType regid` to be able to contain exactly the different number of registers we require to distinguish, instead of using a superfluous `int regid`.

```
#define NUM_REGS 16
#define LOG_NUM_REGS
typedef RegIdType int##LOG_NUM_REGS##_t;
RegIdType regid;
```

4.2 Black-Box Component Substitution

To allow the interval model to act as a drop-in replacement for a processor core model, we modify the basic interval model as presented in [3], which requires completely synchronous knowledge of future downstream execution latencies across the memory hierarchy in order to compute stalls. In the original model [3] requests to the cache block were performed at the head of the new window as:

```
if(inst.isMemOp){
  int miss_latency = Dcache_and_DTLB_access(inst);
}
```

As we must preserve the ability for the cache hierarchy to actually execute concurrently, such synchronous blocking communication is not just a simulation performance issue but rather makes it impossible to integrate the model into a concurrent simulation environment. To resolve this issue, we introduce request/reply buffers that are populated by the TraceFeeder FSM. Requests are inserted into the appropriate resource queue as they enter the tail of the new window. Requests are tagged with the id of the instruction in the new window to mark the instruction as being complete by the time it reaches the new window head. This request interface with a generic queue between the interval model and outside subsystems allows for integration with an external timing simulation framework.

This snippet shows code similar to our request/reply interface:

```
if(inst.isMemOp){
  memRequestQ_enq(inst, tag);
}

while(!memReplyQ_empty()){
  memResp = memReplyQ_deq();
  newWin[memResp.tag] = done;
}
```

4.3 Bounded Memory

All the data structures of our FPGA-friendly interval model had to have an explicitly defined memory size. For example, the buffers between our core model and the cache could not be an unbounded C++ Vector type.

```
memRequestQ.pushBack(inst);
```

Instead, the `memRequestQ` must have a defined size. Since there cannot be more memory requests outstanding than the number of instructions in the New Window, we max size the `memRequestQ` to be the size of the New Window.

```
#define ROB_LENGTH 128
#define NEW_WINDOW_SIZE ROB_LENGTH
#define REQUEST_Q_SIZE NEW_WINDOW_SIZE

int##InstWidth##_t memRequestQ[REQUEST_Q_SIZE];
```

4.4 Common Path Resource Prioritization

Every instruction flowing through the interval model's pipeline eventually enters into the Old Window, making the depen-

dency checks that calculate the issue time for every instruction on the common-case path. We optimize this dependency check using a auxiliary register map that stores the issue time of the last instruction to write to a given register, and a store buffer suitable for computing ST-LD issue time dependencies. By doing a direct lookup into the register map we can efficiently compute the input dependencies issue time. By using these tables we can avoid a full ROB-deep search through the Old Window on the common case. A detailed micro-architecture diagram of this revised model is presented in Figure 3.

The naive way to implement the dependency checks would have been to have a full, O(n), backwards search to find a dependency on an Old Window instruction.

```
int depIssueTime=0;
int i=oldwin.tail;
for(int j=0; j<oldwin.size; j++) {
  if(dependency(instr,oldwin.instrs[i])) {
    depIssueTime = oldwin.instrs[i].issueTime;
    break;
  }

  i= (i>0) ? (i-1) : (OLD_WINDOW_LENGTH-1);
}
```

Our implementation added a register map array to store the issue times and the common path was reduced to simply indexing into this array.

```
IssueTimeType issuetimes[NUM_REGS];

...

int depIssueTime=0
for(int i=0; i<instr.numDependencies; i++) {
 int issuet = issuetimes[instr.dependencies[i]];
 if(depIssueTime > issuet)
   depIssueTime = issuet;
}
```

5. Results

Using the modified FPGA-friendly model presented above, we synthesized our interval model using a modern commercial high-level synthesis tool from 'Vendor A' to translate from C to Verilog. Our interval model was written in 931 lines of code. The generated RTL was verified using cosimulation with the sequential software model and an industry standard synthesis tool was used to synthesize the Verilog to a Xilinx Virtex-5 part. Synthesis results are shown in Fig. 4. As the intention of high-level synthesis of a high-level model is to use the scarce FPGA resources as efficiently as possible, we also synthesize a simple in-order MIPSv1 32-bit core for comparison [4].

From Figure 4 we see that after the set of FPGA-friendly transformations that were applied to the abstract interval model, the synthesized simulation model is a small multiple larger than the full inorder core while providing the timing model for a complex out-of-order core. This overhead is effectively 2.6x in combinational logic, and 5x in flip-flops/BRAM usage. While these costs are not insignificant, the increased fidelity supported by the model is the tradeoff provided for this footprint. While these costs only include the core pipeline and the interfaces required to communicate with memory hierarchy simulation modules, previous work has demonstrated efficient partitioning of CPU-FPGA hybrid simulators [1].

	Interval	Plasma
LUTs	6091	2319
Flops	3609	717
Block RAMs	16	3
Frequency (MHz)	110	55

Figure 4. Synthesis Results for a Virtex5 LX30

As we did not directly optimize the code for a particular HLS framework, the overall throughput numbers provided by push button synthesis are somewhat poorer. While a high overall frequency can be achieved by the synthesized pipeline, the effective simulation rate requires 63 host cycles to compute an instruction on average. With a 100Mhz clock, this yields a maximum throughput of 1.59 MIPS for cracked x86 uops from a synthetic trace. While we explicitly attempted to structure the code to enable O(1) operations on the common path, a straight-forward push button flow implemented certain operations sequentially. The only loop that was manually marked as being unrolled partially was the store buffer search in the OldWindow which would have required a 30 way search if fully unrolled.

The throughput of the FPGA model is lower, but comparable, to the the same model implemented in software. In particular, our modified interval model has an effective throughput of 2.02 MIPS on a Intel Xeon 5140 2.33Ghz processor. Furthermore, as the interval model presented has a limited FPGA footprint, it may be spatially replicated to increase overall throughput, as well as having tight synchronization with memory hierarchy and network-on-chip FPGA simulators.

Furthermore, from a hand-analysis of the micro-architecture presented, we believe a either a manual implementation of the model directly in Verilog, or manual optimization of the code for a specific HLS tool can yield significantly better results, albeit with less flexibility than an automatically generated model.

6. Conclusion

We have presented a case study of using a high-level synthesis tool to automatically transform a high-level simulation model into an reasonably efficient FPGA simulation model.

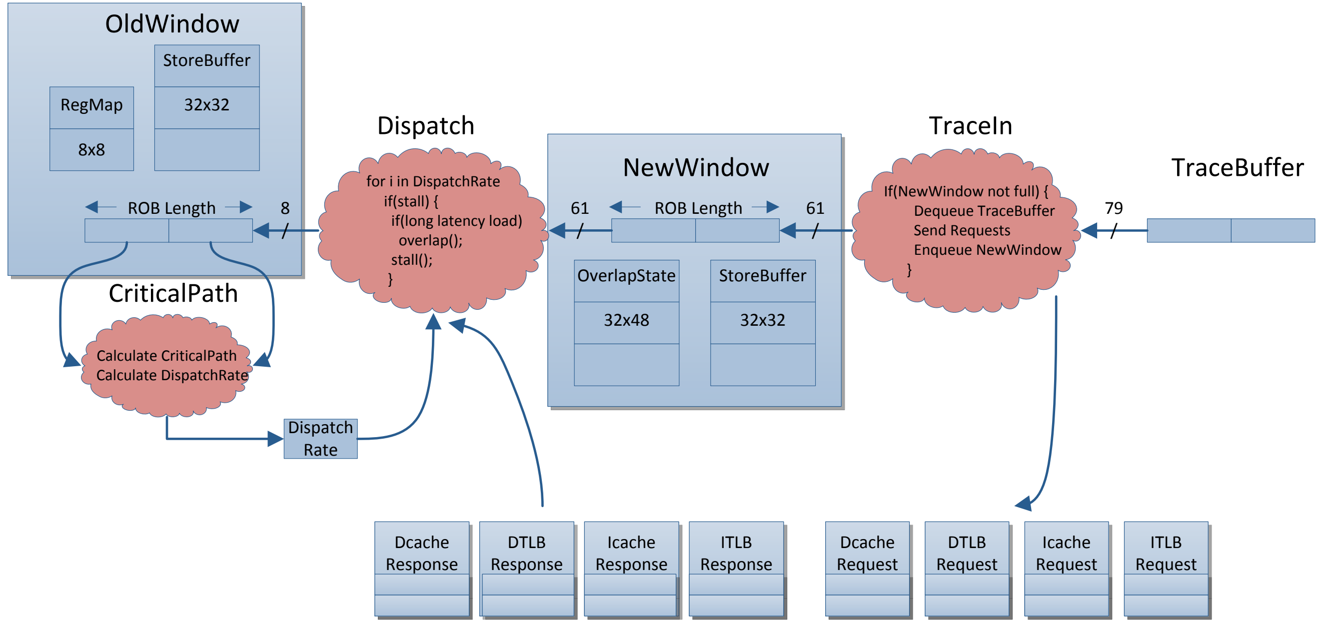

Figure 3. FPGA Optimized Interval Model Micro-architecture

While some work was required to transform the previously proposed interval simulation model to operate in a fully parallel distributed simulation environment, such a transformation was still able to preserve the essential characteristics of the model. The set of FPGA-friendly transformations we have used can still provide for highly productive environment to prototype high-level models suitable for initial placement on FPGA-based simulators.

While our results show there is still room for improvement in high-level synthesis, we can still achieve comparable throughput to a straight software simulation using the exact same model with almost no optimized push-button synthesis flows. By providing a path for transforming a high-level software simulation model into an efficient FPGA model, high-level synthesis for high-level simulation models allows researchers to focus their development effort only on the specific blocks of interest.

References

[1] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, full-system, cycle-accurate simulators. *Proceedings of the 40th Annual International Symposium on Microarchitecture*, Dec. 2007.

[2] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai. A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs. In *Proceedings of International Symposium on Field Programmable Gate Arrays*, Feb. 2008.

[3] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[4] Opencores.org. Plasma. http://opencores.org.

[5] D. Patterson, Arvind, K. Asanović, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, , M. Oskin, J. Rabaey, and J. Wawrzynek. RAMP: Research Accelerator for Multiple Processors. In *Proceedings of Hot Chips 18, Palo Alto, CA*, Aug. 2006.

[6] M. Pellauer, M. Vijayaraghavan, M. Adler, and J. Emer. Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[7] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A Case for FAME: FPGA Architecture Model Execution. In *International Symposium on Computer Architecture*, June 2010.

[8] D. Wang, N. E. Jerger, and J. G. Steffan. DART: Fast and flexible noC simulation using FPGAs. *5th Annual Workshop on Architectural Research Prototyping*, 2010.

[9] J. Woodruff, G. Chadwick, and S. Moore. Cache Tracker: A Key Component for Flexible Many-Core Simulation on FPGAs. *5th Annual Workshop on Architectural Research Prototyping*, 2010.