2015-06-01

Address Space Translation for FPGA Accelerated Simulators

Michael Thaddeus Chamberlain
Brigham Young University - Provo

Address Space Translation for FPGA Accelerated Simulators

Michael Thaddeus Chamberlain

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

David A. Penry, Chair
James K. Archibald
Doran K. Wilde

Department of Electrical and Computer Engineering

Brigham Young University

June 2015

ABSTRACT

Address Space Translation for FPGA Accelerated Simulators

Michael Thaddeus Chamberlain
Department of Electrical and Computer Engineering, BYU
Master of Science

Microarchitectural simulation is needed to help explore the large design space of new computer systems. These simulations are taking increasingly longer amounts of time to run due to the increasing complexity of modern processors. Co-simulation and high level synthesis are promising fields to improve the overall time required for microarchitectural simulators, and can contribute to low design times and fast simulation speeds permitting a larger range of design space exploration. While promising, co-simulation techniques must find effective ways to map the host memory address space to the FPGA memory address space to be able to correctly transfer simulation data between the host and FPGA.

Load relations mapping is a new technique that builds upon existing techniques to provide support for the discovery and translation of runtime memory addresses to their equivalent FPGA memory addresses. This is accomplished by storing object reachability information discovered during a memory profiling run and later using it to recreate an object reachability mapping at runtime. This mapping can be traversed to discover needed memory addresses. We demonstrate how this technique can be used by incorporating it into the FAMEbuilder tool flow. Results show that simulation speed is not reduced and that only a small overhead is required to perform the additional memory initialization at the start of simulation. Area increases are also shown and are limited to near 10% increase on small single core models.

Keywords: Cosimulation, FPGA, Address Space Translation, Load Relations

ACKNOWLEDGMENTS

I would like to thank all of those who have helped me on my educational journey throughout my whole life. This thesis is the culmination of many years of study and hard work and is a direct result of the support I have received from countless friends along the way.

I would like to especially thank Dr. Penry who has guided me throughout the research process. He provided many opportunities to learn and grow and his support was critical in helping me polish and refine my thoughts into a coherent paper.

I would like to thank my friends and family who supported me in working towards my Masters degree and giving me the strength to work through trying times.

Lastly and most importantly I would like to thank my wife Amanda for helping me through it all. She helped me stay motivated when research was progressing slowly and helped me push through the many long hours of writing needed to produce this thesis. She sacrificed many hours by reading through technical jargon to help me polish my writing and rarely complained. She loved me through it all and she is the best support I could ask for.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Microprocessors run the modern world. Processors are in almost every device imaginable and are becoming more and more prevalent, being placed in devices ranging from phones, tablets and computers to items such as cars, watches and heavy machinery. As processors continue to find more and more applications to run on there is a push for processors to run faster and be more energy efficient. Technological barriers, such as thermal limits, prevent designers from being able to rely on frequency scaling and transistor density to improve performance and are causing them to look towards architectural improvements.

There are many components to creating the new generation of processors. Many details such as the number of cores, frequency, cache sizes and protocols, pipelines, power, and area size contribute to the design space of designing a new microprocessor. Computer architects must find ways to evaluate the design space in order to be able to handle the tradeoffs of different design components and to be able to predict the performance of newly designed processors. As the complexity of processors increases it is becoming more and more difficult to quickly and accurately simulate new computer architectures.

Simulators are used to verify the functionality of a design and to give an indication of many performance measurements. The runtime of these simulations has increased with the complexity of processor design and can now easily take over a week to complete on many multi-core designs [1]. Traditionally architects have relied upon the greater processing power of new processors to handle the increasingly complex simulations but increases in processing power have not been able to keep pace with the increasing complexity of computer architecture simulation.

To combat the increasingly large simulation times, many techniques are being developed to decrease the simulation time and permit a greater exploration of the architectural design space. One promising technique is the offloading of parts of the design to field-programmable gate array (FPGA) hardware accelerators [1–3]. This co-simulation uses a host processor to orchestrate the

simulation and FPGAs to run many of the largely parallel components of the simulator. The ability of the FPGAs to run parallel tasks permits them to model these parts of the simulator at much faster rates and can improve the overall simulation time. While co-simulation can improve simulation performance it comes at a cost and has some drawbacks. Designing the HDL needed to model the simulator in hardware can be error prone and slow to write and the increase in design time can outweigh any gains made in the simulation time. To overcome the large design time, high-level synthesis (HLS) techniques can be incorporated to automatically generate needed HDL. When HLS is combined with co-simulation it creates an environment where design time can be similar to that of software-only models but gains the performance benefits of FPGA accelerated code.

While promising, co-simulation and HLS techniques still have their own shortcomings; once an FPGA is programmed its functionality cannot be changed. In order to change runtime behavior a mechanism must be programmed onto the FPGA beforehand to execute that change. Entire segments of code must be dedicated to the loading and offloading of data and must explicitly state the values to be loaded at runtime for proper functionality. This data must be translated from the host address space into the address space of the FPGAs to avoid pointer and address corruption across the host and FPGA interface. Such a need can arise when the host receives runtime input for the simulation that needs to be placed onto the FPGA as simulation begins.

This thesis contributes a solution to translating addresses and pointers between the host and FPGA address spaces. It builds upon object reachability techniques similar to those found in garbage collectors, which create mappings of which objects are reachable from a base set of root objects. We extend this concept to form a new technique called *load relations mapping* which provides the ability to translate a profile run address(FPGA) to its equivalent runtime host address by making use of load instructions that relate objects to each other in memory. By storing reachability information for all objects during a profiling run the same mapping can be recreated at runtime permitting the discovery of runtime addresses directly from the stored mapping. This provides the ability to copy the host simulation memory state and translate and transplant it into the FPGA memory. This technique does so without the need of hand-generated user code to manually map host processor memory values to their equivalent FPGA memory locations. It also provides a means to offload specified end of simulation FPGA memory values and load them into the host memory by mapping the FPGA memory addresses to the runtime host addresses.

CHAPTER 2. BACKGROUND AND RELATED WORKS

The work described within this thesis is designed to augment the abilities of computer architecture simulators that operate within a co-simulation environment. This chapter describes the current research attempts to improve the rate at which computer architectures can be simulated and explored and provides details that lay the framework in which a load relations mapping may be applied.

2.1 HLS

The idea of working in a higher level of abstraction to improve design time and overall productivity of large-scale production has existed since the 1950s. One of the first levels of abstraction was the move from machine code to assembly code which helped programmers reduce errors in their code and finish larger products in faster times. This trend of raising abstraction levels has continued to the present day.

High level synthesis (HLS) is one of the latest efforts to reduce design time and improve overall efficiency by working at a more abstract level of design. HLS entails using a higher level language, such as C, C++, SystemC, BlueSpec and others to represent an abstract level of hardware that can be synthesized into a working hardware design. These high level languages can often be explored automatically by HLS tools to produce optimized designs based on area, speed and power, saving much effort through the use of automated design tools. The designs generated by HLS tool are not always as optimal as hand created designs but generally are comparable and sufficient to meet design specifications.

HLS tools are not capable of converting just any code into a hardware representation but are instead limited to a synthesizable subset of the language being used. The subset of the language is often limited to simple control flow and data structures that easily correlate to certain hardware designs, often consisting of only basic data structures. Many high level of abstraction data struc-

tures cannot be easily synthesized and are often not included in the synthesizable subset of a given language. For example, within the SystemC language simple data types such as floats and pointers are for the most part not part of the SystemC synthesizable subset [4, 5] and are not supported in many HLS tools. Ongoing research is being done to help improve the amount of code that can be synthesized permitting more flexibility in the code being generated and permitting larger portions of code to be synthesized and placed into hardware [6, 7].

2.2 Co-Simulation

Simulation is used to aid in determining the functionality and performance of computer architectures. It provides meaningful measurements that aid in the design of next generation processors. The speed of simulators can no longer keep pace with the complex designs of modern processors and methods must be developed to increase the rate of simulation.

Co-simulation, or hybrid simulation, combines a host processor and hardware acceleration through the use of FPGAs to increase the speed of simulation. Processors are built from hardware logic which performs many tasks in parallel. Many single cycle operations, such as cache reads, take multiple cycles to simulate in software because software cannot simulate the tasks in parallel. Instead, software must sequentially simulate each parallel part of a cache read over multiple clock cycles. These tasks are much better suited to be run in hardware and can be placed on the FPGA to run in simulation, potentially increasing the speed of simulation by taking advantage of and exploiting parallelism.

Additionally co-simulation requires the development of the hardware HDL and the communication interface between the hardware and software. Hardware design can be a slow and error-prone process when compared to code design in a higher level language. Long design times discourage many from attempting simulation in hardware because the long design time can negate all performance gains. Different techniques to reduce the design time include using HLS [6, 7] to generate the HDL, modularizing the design components [3, 8, 9], and creating libraries to aid in hardware design [9–12]. Additionally the communication between the host and the FPGAs must be developed and increases the amount of work required to design a co-simulation tool. It has been shown that reducing the frequency of communication is important to prevent the communication overhead from limiting performance [9, 13, 14]. Additionally, poorly designed simulators may see

no speedups at all due to the added overhead of slow communication between the host and the hardware.

2.2.1 Co-Simulation Frameworks

There are a variety of project frameworks that have been developed over the years to aid in the design and execution of co-simulation. The abilities and goals of each project vary as well as their ability to effectively improve simulation performance. A discussion of various tool and projects will be mentioned here but does not represent a comprehensive list since there are many such tools that have been created.

LEAP

LEAP [8, 10] attempts to bridge the gap between platform support for hybrid software and FPGA applications to the support found in most software programming environments. Most software programmers are accustomed to having rich memory management libraries, portable communication protocols, and block and character devices. The realm of FPGA and software co-simulation lacks many of these tools and leaves the user working on low level constructs, permitting LEAP to bridge many of these gaps. LEAP creates a platform that aids in the design of other simulators, simplifies the design and lowers the design time.

LEAP also extends the support it offers for communication protocols to include automatic memory management in the form of Scratchpads [8]. These Scratchpads are designed to mimic the style of automatic memory management found in software programming and are meant to be a plug-in replacement to FPGA RAM blocks. They provide access to an expandable memory structure that has various levels of memory ranging from on-board FPGA local memories up to software supported host system memory. This memory management protocol allows memory to be modeled and managed at a much higher level of abstraction and permits designers to focus less on implementation specifics for memory hierarchy.

Asim and HAsim

Asim [3] is a performance model framework designed to cope with complex microprocessors. It recognizes that longevity and usefulness of models is hard to achieve and that there are not many structured ways to achieve it. To solve these structural challenges they designed a framework for creating models using reusable modules. The modularity gives a more software like feel to developing performance models by having interchangeable modules that can be instantiated and reused. This permits increased productivity and gives more confidence in the robustness of the software component itself due to reuse of previously tested and validated design modules.

HAsim [15] is a product of the Massachusetts Institute of Technology and Intel Corporation. It builds upon the Asim software model to create a hybrid simulator that models a given architecture. Using the modules from the Asim library, it develops a model that uses FPGAs to accelerate many of the submodules used to build the architecture. It supports multiple platforms and hides the details of communication between host and FPGA thereby permitting more effort on the architecture design and less effort on communication details.

ProtoFlex

ProtoFlex [9, 16, 17] is an FPGA-accelerated hybrid functional simulator designed for multiprocessor hardware and software research. It also modularizes components to build a system piece-wise and then co-simulates them with FPGA hardware acceleration. It uses a hierarchy of execution components to dynamically transplant program execution to the execution component best suited for the given task at hand.

A technique that sets this system apart from others is its transplant capabilities. Components, such as a CPU, can be partitioned into small core sets of frequent behaviors and more complex and less common behaviors. Complete delegation of these tasks to either the FPGA or host can result in slow simulation or complex development and is solved by transplanting, which is re-assigning component execution between FPGA or software simulation at runtime. For most behaviors the FPGA can handle things remarkably well, but occasionally discovers situations ill-suited for FPGAs implementation. When a complex task that is ill suited for FPGAs or an unimplemented behavior occurs a transplant can take place to move the responsibility for those com-

putations over to the software, or another better suited resource. The results of transplanting are a simplified FPGA design that does not need to handle all possibilities and a flexible system that still can take advantage of hardware acceleration in the common case.

The communication overhead of performing a transplant can be very high and potentially bottleneck performance. This overhead is a result of transferring data from FPGA to software and the amount of time it takes the software to perform the given task. To reduce communication penalty a hierarchy, similar to that of a cache memory hierarchy, is developed to reduce the communication latency. An embedded processor or similar kernel can be placed before the software to handle many of the needed behaviors. This kernel is slower than the primary stage FPGA hardware but is able to handle many more instructions and behaviors and pay a much smaller transplanting penalty.

RAMP Gold

RAMP Gold [1, 18] is an FPGA-based architecture simulator developed at the UC Berkeley Parallel Computing Lab and aims to allow an early design-space exploration of manycore systems. It separates the functional and timing models and simulates them separately forming a high-throughput cycle accurate full-system simulator that is capable of booting real operating systems. Ramp Gold can run on a single FPGA board and simulate a 64-core shared memory system achieving up to two orders of magnitude in speedup and is an implementation of their FPGA Architecture Model Execution (FAME) model.

FAbRIC

The FAbRIC (FPGA Research Infrastructure Cloud) project is a tool system being put together by UT Austin and collaborators that aims to provide open access to FPGA hardware accelerated simulation research. It is based on the Convey MX system and has support from Altera, Xilinx, Intel, BlueSpec, and Impulse Accelerated Technologies in contributions of FPGAs, compilers, and other product support. FAbRIC hopes to be able to port its own reconfigurable hardware systems and that of other research groups into the FAbRIC tool system to provide additional sup-

port to those wanting to use co-simulation environments for their own research without the need of building their own system.

2.2.2 FAMEbuilder

The techniques that will be described in chapter 3 are designed with the intent of enhancing the capabilities of the FAMEbuilder tool flow. FAMEbuilder [6, 13] is an HLS backed co-simulation project that seeks to automatically synthesize portions of a structural simulator onto reconfigurable hardware. It takes a structural simulator written in the SystemC language along with a user provided partitioning specification file and synthesizes the RTL description of the partitioned hardware. This RTL synthesis is done automatically alongside the creation of the required software code to run the hardware accelerated simulator. This creates a software simulator with communication wrappers to manage the offloading of the partitioned hardware-accelerated portions of the simulator.

To determine how to best synthesize the hardware partition, the FAMEbuilder tool first begins a profiling run of the target structural simulator. Here the inputted structural simulator is examined in close detail to determine how to best accelerate the code specified in the partitioning file. This profiling run analyzes memory contents from program start to the end of the profiling run in order to determine hardware resource generation and initial memory values to populate the resources with.

Synthesis in the FAMEbuilder tool occurs after elaboration, and as a result causes data structures to be created in the profiling address space. All loads and stores from the original SystemC simulator code are represented in the profiling address space in the hardware implementation. This profiling address space is different than the address space that exists at runtime. During simulation runtime, the FPGAs must take over at the same point in time and understand how the runtime data structures correlate with those that existed in the earlier synthesis run. A mechanism is put in place to transfer control to the FPGAs at this point in time and to be able to transfer any needed data to the FPGAs. The data being transferred between the host processor and the FPGAs must not contain any data that is dependent on the address space being used, such as pointers, in order for it to function properly. To be capable of transferring pointers and addresses between the FPGAs

and the host a mechanism must be put in place to translate between the two address spaces. This mechanism is through the use of load relations mappings as will be described in 3.

CHAPTER 3. RELATIONSHIPS BETWEEN OBJECTS IN MEMORY

Objects in memory usually do not retain the same address from run to run. There are a variety of reasons why this occurs including relocation, address space layout randomization, and dynamic memory allocation.

Relocation [19] is the process of assigning load addresses to various segments of a program and is a process often performed by the linker at link time. Relocation involves moving all segments of code into a single executable and marking segments that have addresses that need updating. The marked addresses are later modified so that they point to the correct runtime addresses based on data stored in the relocation table. This causes any modification of the code base, and conversely any recompilation of the program, to alter the location of data relative to other parts of the program executable.

Address space layout randomization (ASLR) [20] is a security technique that involves randomizing the location of objects in memory in each run. ASLR was first developed for the Linux kernel in late 2001 and was created to combat a large number of exploits attackers could use to jump into an exploited function. It finds and moves key addresses in the code such as the base address of the executable, stack, heap and libraries to discourage these exploits. This also means that the location of data that could previously be predicted can no longer be found as easily. Not only does this make it hard for attackers to locate exploitable locations but it also makes it hard for programmers to specify address locations and memory layouts from run to run.

Dynamic memory allocation is used to allocate memory at runtime for new data. Allocated dynamic memories are assigned a random address from the heap, which cannot be determined beforehand, from a range of free address spaces. This makes it impossible to know the address of dynamic memory until it is created during program execution.

All these concepts lead to the addresses changing between an FPGA being programmed during a profiling run and host addresses used at runtime. Hand-coded efforts to map these two

address spaces can be time consuming and tedious and are not conducive to efficient programming. Addresses needing to be loaded onto an FPGA after to start of simulation need to be translated to match the address space currently programmed on the FPGA. These mappings of host runtime addresses to FPGA addresses can be difficult to find since an object's relative location to all other objects can change from run to run and there currently are no strong techniques to map the two addresses spaces to each other.

3.1 Roots, Pointers and Reachability

Even though segments of an executing program are placed in randomized locations there still must exist a reliable means to find these pieces of data. The ability to find data at runtime is not a new concept and is performed by many different applications.

All programs consist of a base set of root objects. These root objects are a list of all objects that are accessible outside of the heap [21] and are object references the program can access directly, without going through another object. These objects form the base from which all other objects are accessed within a program. All other objects created throughout the life of a program are referenced directly or indirectly through a root object.

A program object is considered reachable if it is referenced by at least one other reachable object or belongs to the root set of objects. Therefore any reachable object that is not a root object can be reached either by another reachable object or is referenced by a root object. All reachable objects can trace references to them back to a root set object because all objects are reachable from the root set.

The references between objects that link them together and determine reachability are pointers, or addresses stored in one object that give the location of the object to which they refer. Through these references a program is able to grow and shrink in size and still maintain an up-to-date knowledge of the location of all objects that pertain to the current program execution. As new memory is allocated a reference to that object is created and stored in another reachable object, linking the new object into the program. These references can be traversed to navigate from one object to another and to locate data as it is needed during the program execution.

Garbage collection [22,23] is a practical implementation of the concept of reachability and exploits the fact that all objects map back to a root object or are no longer an active part of the

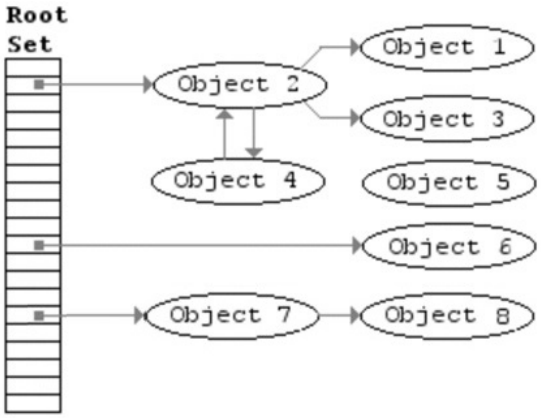

Figure 3.1: Objects and Pointers: A sample of how objects relate to each other in memory. Similar to how garbage collection works, if an object does not relate back to the root set it is no longer an active part of the program.

program. Garbage collectors are used to free up allocated memory space that is determined to no longer be used within a program so that the memory can be used for the allocation of other objects. Trace garbage collectors function by examining the set of root objects and finding all objects referenced and reachable by the root objects. All other objects not referenced are then collected and their memory is freed for further use.

Program pointers, used to determine object reachability, often exhibit the property where they point to the same object in all executions of a program. Many pointers are written into the program executable to point to specific data structures and are stored in specific locations, causing them to always refer to the same object in memory, regardless of its location. This is beneficial and needed since objects are randomly assigned memory locations at runtime and there must exist a reliable method to discover those objects.

Not only do many of the pointers point to the same object each run, but they also are stored in the same relative location within the object. Objects have a fixed shape in memory which leads to fixed locations of specific pieces of data, such as pointers to other objects. Such fixed locations are what permits the program to acquire specified data as it is needed. This means that in any given execution of a program if the location of an object holding a pointer is known and the relative position of the pointer within the object is known then that pointer can be accessed.

A program executable understands that to obtain access to a certain object in memory it can traverse a series of references to find the desired data. The object labeled `Object 1` in

figure 3.1 has a known reference to it from `Object 2`. Therefore anytime `Object 1` is needed it can be accessed through `Object 2`. Additionally `Object 2` is known to be referenced from a specific object in the root set and can always be found through that root object. By establishing and mapping these relationships a general path can be created to discover the location of all reachable objects in memory.

3.2 Mapping Objects by Relationships

Most methods of referring to objects in memory consist of using address locations and offsets to locate data. This addressing matches with the design of memories where data is stored sequentially and is mapped to specific locations. This causes objects to be located by an address location inside a memory unit or an offset from another object in memory. In order to access any data in this structure its address must first be obtained.

Using addresses to map objects in memory places a limitation on what objects can be referred by only permitting objects that have a known address. Addresses found in profiling runs cannot be used to describe runtime addresses because these addresses will change and are unknown during profiling. Any profiling addresses discovered are only valid within their address space and become invalid during runtime execution causing any mapping of data through use of their addresses to to be transferable between different executions of a program. Limitations such as this can be overcome by building a new memory mapping that references objects not by their addresses but by their relationship to other objects in memory.

A mapping based by relationships trades addresses and offsets for relationships to other objects in memory. This causes the new memory model to look similar to that of figure 3.1 where there exists a base set of root objects through which all other objects are referenced. To access the data in an object the object must be located by traversing the list of pointers that link the objects together. Addresses are needed to actually reference the memory values of these objects because they are in memory, but a representation of how objects relate to and reference each other provides a different method of mapping objects in memory.

A relationship mapping has the benefit of not needing an address to describe how an object can be accessed. It describes the acquisition of data by describing the traversal from one object to another until the target object is reached. This allows the representation of not yet allocated

memories by abstracting away the missing address and only using the relationship between the memory and the object that stores the pointer to it.

A form of relationship mapping is exhibited in part by how a linker represents programs as they are being linked together [19, 24, 25]. During the linking process symbols are used to refer to many objects that are as yet unknown to the linker. These can be external variables or references that are not yet resolved. Even though there is not a physical mapping to the data being represented, a symbolic mapping is created that describes how the unknown objects fit within the program. When the references become available the linker updates the symbols to contain the correct information.

Dynamic linking with dynamic link libraries also has the same issue of unknown references that must be represented before the reference can be resolved. Until a program begins execution the location of dynamic libraries is unknown to the program. While it does not know the memory address of these libraries, it retains enough information to represent and access them once they are loaded and become available at runtime.

These relationship mappings of objects are not capable of retrieving the referenced data until the data is available, but they contain information regarding how the object will be located. This data can be recorded and stored as a way of describing unknown memory locations when they lack an address and can be used later to access these memories once they receive an address in memory. This principle can be applied to memories that lack knowledge of their runtime addresses at compilation but have enough information to describe how their address will be obtained at runtime. Additionally this can be used to facilitate the mapping of one memory state to another, permitting the transplanting of simulation state between host and FPGAs.

3.3 Load Relations

Two new terms are defined in this paper and are used to describe an implementation of the relationship mappings described in the previous section.

The term *load relation* describes the relationship that exists between objects that links them together within this relationship mapping. It derives its name from the code's load instructions that are used to discover object addresses in many low level code representations. It is through these

load instructions that addresses and pointers are accessed to locate other objects in memory and what inspired the creation of this object mapping.

The term *load relations mapping* describes an expanded form of the relationship mapping discussed in the previous section. This mapping describes the relationship between objects in memory independent of the memory address of objects but also includes information necessary to translate this mapping back into a memory address. A load relations mapping consists of objects and load relations that link these objects together. Additional information stored includes profile run addresses and offsets that permit this mapping to translate back into an address.

The goal in the creation of a load relations mapping is to gain the ability to discover the runtime address of profiled simulation state memory for use in FPGA memory address translation. As will be shown in future sections, the information stored in a load relations mapping can be used to translate addresses from one program execution address space into another, specifically from the profile run addresses to runtime addresses. The ability to translate addresses permits simulation state addresses to be stored as they are discovered during profiling and later to be translated to provide access to the values used in the runtime execution. Runtime access to simulation state data through use of profile run addresses provides the ability to perform the runtime memory mapping and state transplanting to FPGA memories within a co-simulation framework.

3.3.1 Overview of the Load Relations Mapping Process

Load relations mappings are static data structures and do not change automatically as new objects are created and destroyed, causing a single mapping created during the profiling run to only be valid at a single point in time in the program execution. In order for the mapping to correctly map profiled addresses to runtime addresses it must be insured that the creation and use of the mapping are performed at equivalent times during profiling and runtime execution.

Once a specified point in the program is reached during the profiling run, a load relations mapping is created. This process consists of identifying all program root objects and load relations that exist in the program and creating a load relations mapping based on that data. After creating the mapping it is stored into the program executable for runtime use. During runtime the program must reach the same point in execution for which the load relations mapping was created during the profiling run. Once this point is reached the mapping can be used to locate runtime objects

Table 3.1: Load Relations Mapping Process Overview: Load relations mappings require equivalent memory structures when created during the profiling run and used during runtime. This requires that the creation and use of the mapping be performed at equivalent points in the program execution.

Profile Run	Runtime
Stop at marked point in program	Stop at marked point in program
Identify root objects	Identify addresses of interest
Identify load relations	Traverse load relations mapping
Create load relations mapping	Translate addresses
Store mapping	Transplant runtime simulation state to FPGAs

in memory. All objects mapped will exist at this point in the program, including simulation state, which permits the acquisition of runtime state memory values through this mapping.

This process of creating a load relations mapping during a profiling run and traversing the mapping at runtime to provide runtime addresses will be described in the following sections.

3.4 Creating Load Relations Mappings

There are two main components to discovering and creating a load relations mapping for flagged objects: discovering the root objects and discovering the load relations.

Root objects are objects that form the roots of a load relations mapping graph. What sets them apart from other objects in memory is that these are objects whose address can be found at the beginning of runtime execution and provide a starting point to compute the address of other objects whose runtime address is not as readily accessible. These objects are an inherent part of the program and must be collected to form the roots of the load relations mapping.

Load relations exist when one object holds a reference to another object. These references can be discovered by examining low level descriptions of the program code and finding all load instructions. These load instructions can then be put through a memory analysis tool to determine if a load of an address is being performed or if it is a load of some other type of value. Only loads of address need to be recorded because these are the loads that represent a load relation between two objects. These loads are described as a load relation and consist of the load effective address and the address being loaded.

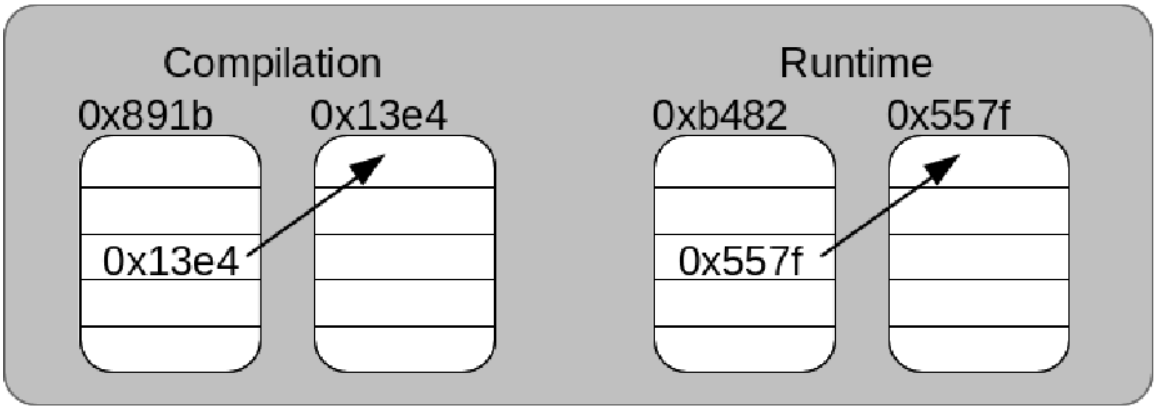

Figure 3.2: Load Relation: Even when memory addresses change between program executions the load relationship that links two objects remain constant.

A record of each load relation is made, initially containing the load effective address and the loaded address. Once a full load relations mapping is created the mapping is capable of abstracting away the memory addresses, but until it has been fully created it is necessary to retain address information to be able to link objects together and to distinguish one object from another.

After all root objects and load relations are gathered a load relations mapping can be created. This is accomplished by creating a graph of nodes and directed edges where nodes represent objects and edges represent a load relation linking the objects together. A node is created for each root object and also for each load relation. A load relation relates two objects to each other and requires that each object be represented as a node in the graph. Only the referenced object in a load relation needs to be added as a node to the graph because the other object will either be a root object or the referenced object by another load relation and will be added by either of those two cases. The address of all root objects and the load address from the load relations are used to distinguish nodes from each other. After all nodes are created edges are added for each load relation linking nodes to each other.

Through this mapping every object that exists within a program will either map to or be a root object. Such a claim is the same argument made in the creation and execution of garbage collectors; Any object not mapping back to a root object in garbage collection is considered to be garbage and is collected so that the memory can be reclaimed and freed for future use. Using the same logical argument, only objects that are alive within a program will be mapped into the load relations mapping.

A load relations mapping, when fully populated, has the property of providing all the needed information to be able to map any program profiled address to its equivalent runtime address. The load relations mapping in the currently described form has the ability to describe how an object can be discovered in memory but lacks the ability to perform the full address mapping without additional data being added to the mapping.

Additional Data Needed to Provide Address Mapping

The load relations map must include more than just the relationships between objects to be capable of translating addresses at runtime. The first piece of additional information needed is the base address of each object found during profiling. The base address is equivalent to the load address of a load relation but needs to be stored separately due to the changing nature of addresses in a load relation. The load relation is a pointer to another object, at runtime the pointer will be changed and updated to match the runtime location of the object it points to. Creating a separate copy of the profiling address allows us to retain profile run information even in the changed runtime environment. The profile run address is also the same address used to distinguish nodes from each other as described previously and is recorded as part of building the load relations graph.

Another piece of information needed is an offset value between the base address of an object and any load relations it contains. The contents of an object do not change between executions of a program, permitting an offset to be stored representing where within the object the load relation originates from. The offset refers to the distance in memory from the base address of the object to the load relation's effective address. Regardless of where in memory an object is placed at runtime, the offset provides the ability to access the load relation it represents from the object that holds it. This offset is stored in the edge of the load relation that it represents and provides the needed information to be able to traverse in memory between the two objects in the load relation.

Lastly the mapping needs to contain additional information to obtain the runtime address of any root objects. For global object roots a pointer to the global object can be stored. Global pointers are updated automatically for each run and can provide an initial starting base address. Root objects are specific to the environment being worked in and other classes of root objects may require other methods to obtain their base addresses at runtime. For example, SystemC root objects can store the name of the object it represents and can retrieve a pointer to the SystemC object at

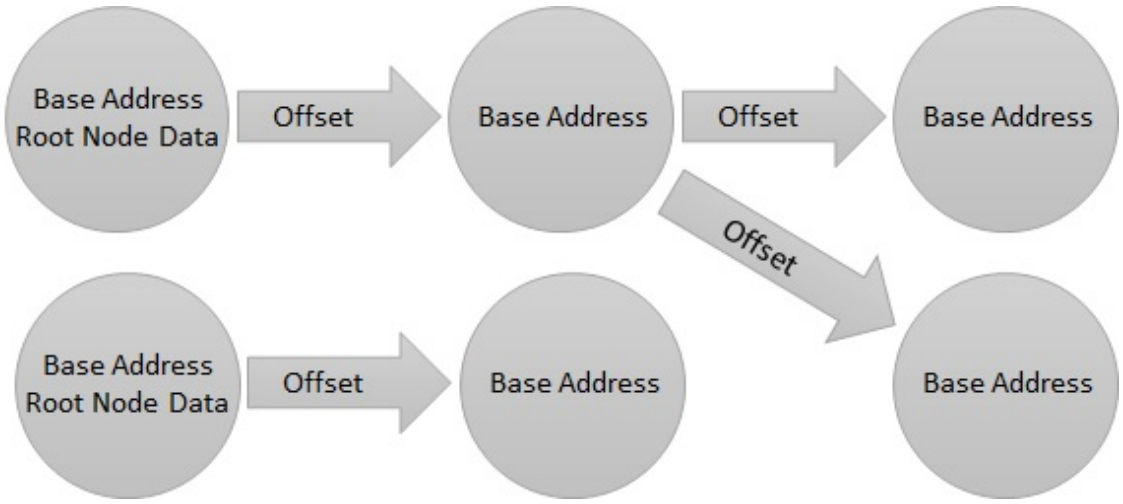

Figure 3.3: Load Relation Mapping Graph: A sample load relations mapping graph showing nodes representing objects and edges representing load relations. The data stored in each node and edge is also depicted.

runtime by supplying the stored name to the built in SystemC function `sc_find_object(string name)`. The root object specific data is stored inside the root object nodes and causes root object nodes to have additional data over that of non root object nodes created through load relations.

3.5 Map Traversal

The load relations mapping can be traversed at runtime to discover how a target address is computed. The first step is to determine which node in the load relations mapping represents the target address. The target node is found by comparing the target address to the base address stored in each node. The closest lower base address is the node that is needed because of how objects are mapped in memory. All of an object's data is placed higher in memory than the object's base address which permits a positive offset from the base address to provide access to all data stored in the object. Even if the closest base address is a higher address than the target address the closest lower address must be used. A higher base address points to a different object in memory and may not be located in the same relative position to the target address at runtime and cannot be used.

Once the target node is determined, an offset is stored indicating where the target address is located within the target object. This offset provides the ability to determine the target's memory address once the address of the object has been determined. Dynamic memory addresses and stack

allocated objects cannot be computed until the memory has been allocated and an address assigned, but for global values and root objects this can be done at any point because the object address is already determined at the start of runtime.

Once the offset is stored a path must be found that leads back to a root node. The path is found by traversing through parent nodes until a root node is reached. The parent nodes are any nodes that have an edge leading towards the current node. The path that is found from a root node to the starting node represents a chain of load relations that can later be used to determine the runtime address of the target object and then the target address.

These chains of load relations can be traversed at runtime by using the information stored in the graph edges. The first node in any load relations chain will always be a root object, providing the means to obtain a starting runtime base address. Once the base address is determined, the offset stored in the edge leading to the next object in the chain is added to the base address yielding the memory location that stores the reference to the next object in the chain. This effective address is dereferenced and provides the base address of the next object in the chain.

The process of following the edges to new objects and using the base address and the offset to create a new load effective address can be repeated until the last object has been loaded. Once the last object in the chain has been reached, the runtime address being computed can be calculated by using the offset stored when the target object was initially found. The stored offset is added to the runtime base address of the final object in the load relations chain and yields the target runtime address.

Traversing the load relations map can be repeated in this manner for every address that needs to be translated from its profile run address to its runtime address. A complete load relations mapping has the information necessary to translate any address that pertains to the execution of the mapped program.

Additionally the load relations mapping can translate any runtime address back into its equivalent profile run address. This backward translation is done by performing the map traversal described previously from the root nodes to all other nodes in the graph. As a node is entered for the first time its runtime address is recorded alongside the previously stored profile run address until all nodes have received an updated runtime address. The runtime address being translated is then compared to every node within the load relations mapping to determine which node has

the closest lower bounded runtime base address. This node represents the object that contains the address being translated. Once the correct node is found the map no longer needs to be traversed and the profile run address can be computed. The offset is again computed between the runtime base address of the node and the address being translated. This offset is then added to the stored profile run base address of the node yielding the profile run address of the address being translated.

3.6 Load Relation Chains for Sparse Applications

The full load relations mapping can be large for large programs and might be more than what is needed for many applications. The number of nodes is the total number of reachable objects in the program, including global variables and data placed on the stack and heap. The number of edges is the number of unique loads of addresses performed in the program. Out of all the nodes and edges that exist in this mapping, only those that belong to a load relations chain pertaining to addresses of interest are needed to determine those addresses. Often only a small number of addresses will need to be computed or translated and storing the entire mapping would waste space since much of the map will not be used to perform these operations.

With the understanding that only a subset of the load relations mapping may be used in many applications a second approach is discussed that permits only a portion of the mapping to be stored. When all addresses are marked and known that need to be translated through a load relations mapping, it is possible to create only the load relations chains that correlate to those addresses. This optimization can provide the potential for faster address computation and less memory usage for sparse applications. The data stored in each node and edge will remain the same but the number of nodes and edges used can be much lower. Using load relation chains does permit the duplication of nodes and edges creating the potential for increased memory usage in non-sparse applications. For this reason it is not always a more efficient method and must be evaluated to determine when it is proper to implement.

The load relations chains are computed individually for each flagged address and only require that a list of all root objects and load relations be computed. Because nodes from the complete load relations mapping can come from either of these two sets, both will need to be checked to determine what nodes belong in the chain. The first object in the chain is found by finding the closest lower bounded object, in either set of nodes, to the flagged address in question.

Once found the offset of the object from the flagged address is recorded and the node is checked to determine if it is a root object node. Determining root nodes is a trivial step in this setup because all nodes created through the load relations list are non-root objects while all nodes in the root objects list are root objects. If the object is not a root object, the offset from the load relation is computed and stored and the process is repeated using the effective address of the load relation as the new address to compare against. This process continues until a root object node is found, forming a completed load relations chain equivalent to that used in a complete load relations mapping traversal.

A list of all load relations chains can then be stored in place of the full load relations mapping. Each load relations chain can be mapped to the address it correlates to and permits logarithmic runtime lookup of a specified load relations chain. It also saves time by eliminating the need to determine what node an address correlates to since this was computed during profiling. Load relation chains permit quick runtime address computations using less memory space at the expense of flexibility and completeness. If other addresses need to be computed that don't pertain to a stored load relations chain then they cannot be translated because the whole load relations mapping does not exist, only a smaller subset which may not contain all the needed information.

CHAPTER 4. IMPLEMENTATION

This chapter discusses a practical implementation of using load relations mappings to auto generate the software and hardware code needed to perform FPGA memory state initialization and finalization of runtime values. This is accomplished by adding the load relations mapping feature to the FAMEbuilder synthesis tool flow, which as stated in chapter 2 seeks to automatically synthesize portions of a structural simulator onto reconfigurable hardware. FAMEbuilder makes an ideal candidate for the load relations mapping feature because it seeks to offload work onto reconfigurable hardware and extends the range of simulator code that can be offloaded into hardware. In addition, FAMEbuilder already has in place a detailed analysis of memory values that aids in the discovery of load relations and significantly reduces the amount of additional computation needed to create a full load relations mapping.

The goal of adding the load relations mapping feature in the FAMEbuilder project is to permit transplanting of runtime host simulation state to and from FPGA accelerator memories. This goal is accomplished through the creation of a load relations mapping to discover the runtime values of any flagged memories and map then to the profiling run addresses used on the FPGAs. Any pointers contained within these flagged memories are capable of being translated into their appropriate FPGA addresses. Additional hardware and a communication interface are developed to support the memory initialization onto the FPGA.

The following sections provide details about the implementation of the load relations mapping feature in the FAMEbuilder project.

4.1 APIs

The first stage of implementation was determining the set of APIs that the user would have access to in order to use the feature in their simulator code. The goal was to make the feature APIs as simple and non-obtrusive as possible so that it would not be a burden upon the

programmer to implement it into their simulator code. Most of the feature tasks are performed within FAMEbuilder and it was determined that the only task required by the user would be to flag which memory regions should be targeted by the new feature.

The API provides the starting address and size in bytes of the flagged memory region and takes the form of `flagMemory(void* address, int size)`. This function call provides all the information needed from the user to begin using the feature. Each call to the function causes the address and size pair to be added to a master list of all address and size pairs that will be analyzed during the FAMEbuilder tool flow. All other data regarding the simulation of these flagged memory locations is determined and created during the profiling run. All memories flagged by this API are analyzed so that their memory values can be initialized onto the hardware and finalized by offloading the FPGA values back into the host memory. Performing both these tasks causes all flagged memory regions to fulfill goal of the feature by performing both memory initialization and finalization of FPGA memory values.

An additional API, `finalize()`, must also be called but is invisible to the user within the FAMEbuilder tool chain. The `finalize()` function signals the end of simulation and informs the project when to offload all flagged memory values from the FPGAs and place them into the host memory. Within the FAMEbuilder project there exists a similar function call that is used to signal the end of simulation and perform simulator finalization tasks of its own. Rather than create a separate API and require two function calls with equivalent purpose the feature `finalize()` function is integrated as part of the tool chain's finalize function and is kept invisible to the user. In other frameworks it may not be possible to hide this API within the tool chain's functions to indicate when the simulator has finished execution and it may be required to be called separately.

4.2 Resource Recognition and Matching

The next step in the implemented feature is to determine what flagged memory regions are partitioned into the hardware. Not all regions flagged by the user are guaranteed to be in a hardware partition and regions not in a hardware partition do not benefit from this feature and should be ignored. In addition some flagged regions that are contained in a hardware partition may not be allocated a resource. This may occur when a marked region has no reads or writes associated with it which are required to create a usable resource.

To insure that all memory regions that need a hardware resource are given one, their initial memory values are marked as a variable or unknown value. Marking memory regions with unknown values helps the memory analysis performed as part of the FAMEbuilder tool flow to allocate resources to all flagged memory regions that have been partitioned into hardware. This is needed for memory values that are constant within a single program execution but may be dynamic when viewed across multiple program executions. This can occur with program inputs and similar variables that are set at the beginning of program execution and that only change when the input is altered during a future program execution.

Once all resources have been generated, a list is made of all the resources that contain a flagged memory region. This step simplifies all remaining analyses by permitting iteration through only those resources that will be used by the load relations mapping feature. This is accomplished by comparing all addresses contained within the resource to the list of flagged memory regions and the addresses they contain.

4.2.1 Region/Resource Pairs

After all resources containing flagged memory regions have been found they must be matched to the actual region they contain. What differs in this stage from the previous step is that additional work must be performed to determine what portions of flagged memory regions are actually contained within the resource. There is not always a one-to-one mapping of memory regions to resources since a memory region may be split among multiple resources or a resource may contain multiple memory regions.

Since there is no simple mapping of whole memory regions to whole resources a smaller division is created to better model what will actually be represented in the hardware. Memory region to resource pairs are created that map all subsections of flagged memory regions to the resource that contains them. These region-resource pairs represent contiguous memory regions that map to a single resource and better represent the potentially fragmented flagged memory regions to resource matching. They simplify memory initialization by removing the more complicated scenario where memory regions map to multiple resources and remove the additional hardware logic required to handle this situation.

The region-resource pairs are then given a fixed ordering that is maintained in the initialization code in both hardware and software. A fixed ordering is used to simplify the communication between the host and the reconfigurable hardware and also simplifies the memory initialization hardware itself.

As a final step all flagged memory regions that are not given a hardware resource are removed from the remaining analyses. These regions belong to the software partition and do not require any additional work to perform initialization or finalization of memory values. Only those regions that are allocated a hardware resource require additional work at this point in the feature implementation process.

4.3 Load Relations Mapping

Next the load relations mapping must be computed, traversed and stored so that the flagged memory values can be read at runtime. The first step in this process is determining all root objects. In FAMEbuilder the list of root objects consists of global objects and SystemC objects and can be found by iterating over a list of these objects previously discovered and stored by the FAMEbuilder tool chain. In this environment, no root objects are available to hardware partitions that are allocated on the stack unless they are also SystemC objects. Because the roots on the stack are SystemC objects they fit the definition of being reachable directly, and not through other object pointers, through the use of the built in function `sc_find_object()`.

The second part to the load relations mapping is the discovery of all load relations. Because of the LLVM framework and the memory value analysis previously performed by FAMEbuilder this process is straightforward within the FAMEbuilder framework. We iterate through all of the instructions and make note of all load instructions found. The memory value analysis performed previously also makes note of what type of value is being loaded and informs us if the load instruction is performing a load of an address or another type of value. Only loads of addresses are considered since these constitute a load relation and the effective address of the loads and the address they are loading are stored into a sorted map of load relations.

4.3.1 Load Relations Chains

With all of the load relations and root objects available a load relations mapping is ready to be created. Since a list of all flagged memory addresses is provided by the user, a full load relations mapping is not needed to represent this subset of addresses. In place of the full load relations mapping a list of load relation chains is created to save memory and to prevent a complete mapping from needing to be created.

A load relation chain is created and for each of the region-resource pairs previously found. The chain is started by taking the starting address of the region-resource pair and finding the load relation or root object that is the closest lower bound match. In our implementation all load relations, global root objects and SystemC root objects are stored in sorted maps and the `lower_bound()` method is used to find the closest lower bound of each group. The closest of these three is the closest lower bound and the offset between the closest lower bound address and the starting memory address of the region is computed and push back onto a list of offsets.

If the lower bounded object is a load relation the process is repeated again using the effective address of the load in place of the starting address of the associated memory region. This process is repeated by pushing back all load relations offsets to the offsets list and tracing the relations back until a root object is reached.

Once a root object is found the last offset is pushed back onto the list of offsets and the needed information is stored to discover the address of the root object at runtime. For a global root object a pointer to the object is stored that will be updated by the linker at runtime to reflect the new location of the global object. For a SystemC root object its name is stored and is referenced by the `sc_find_object()` method at runtime to retrieve its runtime address.

4.4 Hardware Generation

With the load relations mapping complete and the list of all region-resource pairs available, the next step is to generate the HDL to perform the memory initialization and finalization. Being an HLS tool, FAMEbuilder auto-generates all the HDL for the simulator and requires that this feature alter the generated code to be able to perform these additional functions. Because initialization and finalization code is not part of the simulation, but is a feature that is used before and after

simulation, it is added as a separate module that to ensure that it does not introduce a new critical path into the system that would slow down the simulation time.

Separating the feature code is accomplished by creating a separate communication stream to interface directly with the memory initialization and finalization code and having it talk directly to the memory resources. Each flagged memory resource receives an additional read and write port that is dedicated to the memory initialization and finalization code. These ports become the connection between the added code for the feature and the auto-generated simulator code. The port connections consist of a `req`, `ack`, `address` and `data` signal for both reads and writes.

4.4.1 Finite State Machines

Communication between the host and the simulator memory resources is handled by the addition of two finite state machines (FSM), each responsible for handling different ends of the communication. One state machine, the host communication FSM (comFSM) handles communication between the FPGA and the host while the resource FSM (resFSM) handles communication with each of the individual resources. A small buffer is placed between the two FSMs to handle communication between the two. The separation of these responsibilities between the two FSMs simplifies the code and also reduces the amount of work required by the HDL generator.

Communication FSM

The comFSM handles all communication between the FPGA and the host, and it must understand the communication protocols for the platform it is running on. There are many different mechanisms that can be used to perform this communication, including PCIe, Ethernet, and FPGA board specific methods that each require a unique form of transmitting data packets. For this reason a separate comFSM is designed for each platform supported by FAMEbuilder and the correct version is automatically used for each platform.

While each version of the comFSM communicates uniquely according to the platform it works with, each comFSM maintains the same communication interface with the resFSM. A 128-bit buffer is placed between the two FSMs to facilitate the passing of data and is sized to match the most common size of currently supported communication protocols. Through the use of request

and acknowledge signals data is able to pass through the comFSM to the resFSM in a way that is independent of the platform and communication mechanism used in the system. The buffer is designed to permit one FSM to act as a feeder of data into the internal buffer while the second FSM acts as the consumer of the supplied data.

Resource FSM

The resFSM handles all communication with the memory resources and must be generated for each simulator to match the region-resource pairs discovered previously. Because of the fixed ordering establish during profiling there are no addresses sent to the resFSM in the data packets. Instead the resFSM understands where each byte of data needs to go based on its position and order in the total set of data being communicated.

Each region-resource pair is given its own set of states to handle all initialization and finalization of memory values to and from that resource. For each, initialization and finalization, there exists a state to handle communication protocols with the memory resources and a different state to handle communication protocols between the resFSM and the comFSM. With the fixed ordering determined at region-resource pair generation the initialization states resemble that of figure 4.1. In total there are four states for each region-resource pair and two additional states to handle idling and odd sized memory finalization, creating $4 * N + 2$ states total, where N represents the number of region-resource pairs.

In the resource communication states,shift registers handle the data being read to and from the memory resources and are set up to handle reads and writes on the byte or word level. When handling data on a byte level a word sized buffer is used to store the data until the whole buffer has been used. At this time it then treats the data the same as if it had just accomplished a word sized transaction and shifts it through the 128-bit internal buffer. Once the internal buffer is filled the state transitions into the comFSM communication state until the comFSM has had time to provide new data for writes or consume the data on reads, returning control of the buffer back to the resFSM.

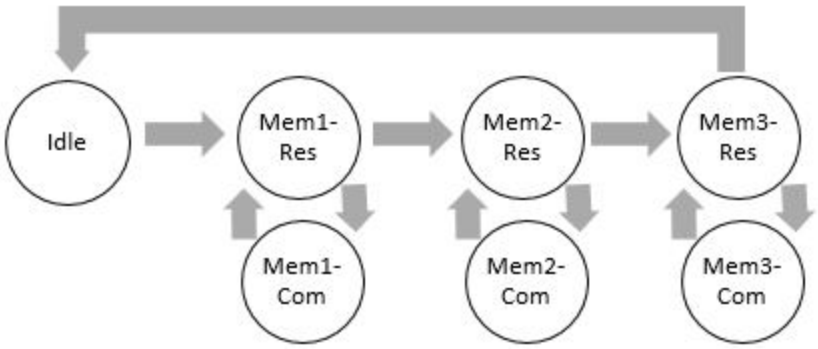

Figure 4.1: Sample resFSM with Three Resource-Region Pairs: A sample resFSM showing the initialization states when there are only three resource-region pairs.

4.5 Runtime

With all of the HDL fully generated all that is needed to complete the feature is to provide runtime support in the software. Runtime support includes storing the load relations chains to the program executable so it can be referenced at runtime, translating the runtime addresses using the load relations chains and creating the runtime APIs to support communication with the FPGAs.

The data in the load relations chains created during profiling must be altered slightly to become compatible with how they must be read at runtime. The variable size offset arrays do not work with the static runtime code that cannot know its size a priori. For this reason the data structure created during the load relation chains generation is altered slightly as it is placed in the program bitcode. Most data remains in the same format but the offset vector is replaced with a pointer to an internally linked array of longs.

For similar reasons the vector of load relations chains must also be converted to a static sized array of structs. Both this change and that of the offsets array are necessary to provide static sized data structures in place of the variable size vectors to conform to the requirements of placing these data structures into the program bitcode.

The runtime data structure closely resemble that of the bitcode data structures and contains an array of load relations chains structs. A long* is used to point to the array of offsets and the size of this array is provided within the load relations chain struct. These provide access to all of the data determined during profiling that is necessary to compute the runtime addresses and values of the flagged memory regions.

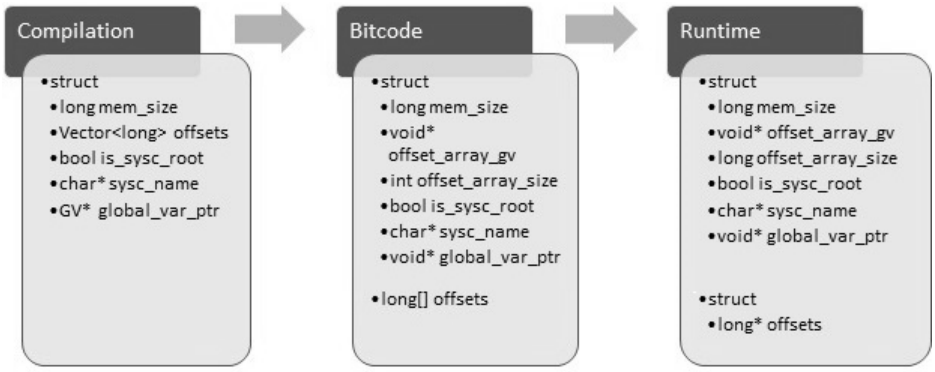

Figure 4.2: Load Relations Chains Data Structures: The dynamic sized vectors used during profiling must be replaced with static sized data structures for the bitcode and the runtime code.

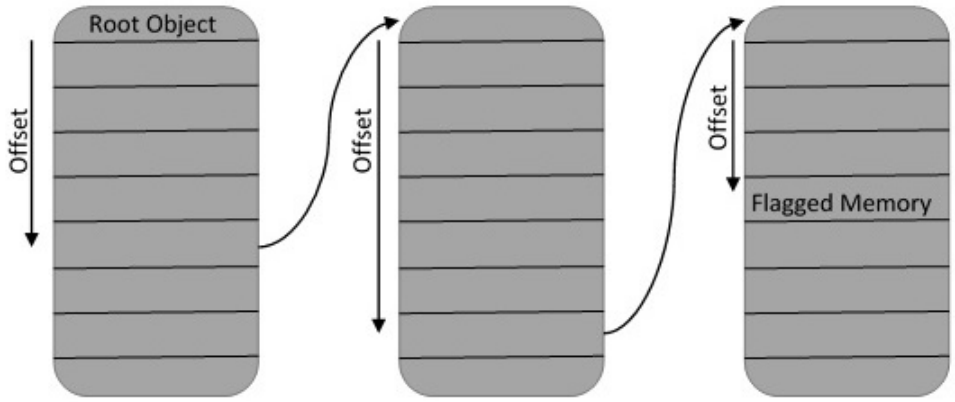

Figure 4.3: Load Relation Chain Traversal: Traversal of a Load Relations Chain starting at the root object base address and traversing through two load relations to arrive at the flagged memory region starting address.

To create the runtime load relations mapping each load relations chain is traversed starting from its root object until it reaches the end of the chain. Traversal begins by finding the address of the root object using the methods described previously and adding the first stored offset to determine the location of the pointer to the next object in memory. The pointer is then dereferenced and the next offset is added to the dereferenced address to obtain the next pointer. This process continues until the last offset in the offsets array is used, which signals the end of the load relations chain. The final value is the starting runtime address of the flagged memory region represented by the load relation chain.

As each runtime address is computed it is again paired with the size of its memory region and the pair is stored in a list of runtime address and size pairs. This list of precomputed values is used in place of the load relations chains to prevent the recomputation of load relations chains each time the runtime address needs to be referenced. The order in which the runtime addresses are computed is also preserved to maintain coherency between the software and hardware since it relies on a fix ordering of data being sent to and from the FPGA.

An initializeMemory and finalizeMemory function are created to help perform the FPGA memory initialization and finalization at the appropriate times. These methods are called automatically by the FAMEbuilder tool chain at the beginning and end of simulation. They function by writing to or reading from a buffer that is populated with all the flagged memory values contained on the FPGA. The buffer is sequentially walked through and each byte of data correlates to a specified flagged memory location according to its location within the buffer. The buffer is used with the appropriate drivers to perform the needed communication between the FPGA and the host system.

These two methods fulfill the runtime goals listed previously and accomplish runtime memory initialization and finalization. Through these methods the runtime simulation state can be mapped and loaded into its proper FPGA memory addresses. Additionally as the simulation finishes these values can be offloaded back into host memory. For program state containing addresses and pointers, these values can be translated to point to the correct data in the different address spaces but require additional support from the user.

CHAPTER 5. EVALUATION

We demonstrate the cost and effectiveness of using load relations mappings to initialize FPGA memory by evaluating its implementation in the FAMEbuilder simulator tool chain.

A simple five stage in-order pipeline that executes the PowerPC instruction set is run through the FAMEbuilder tool chain to simulate a series of processors ranging from a single core processor to up to a 16 core processor. The tool chain is configured to create two simulators for each configuration, one running natively without the load relations mappings feature and one with it enabled. Both simulators are then synthesized and compared to determine the cost of implementing this feature. The regions flagged to perform memory initialization and the offloading of memory values were chosen to perform analysis of stall counts within this simple framework. A total of $3X + 1$ regions were flagged where the $3X$ represents 3 core specific regions being flagged and the 1 region represents a global space variable shared by all cores. The simulations were run on a Xeon E5-2609 quad-core 2.5 GHz processor with 16 GB of memory running RHEL 6.6 and were connected by PCIe to a Pico board containing M-503 modules.

There are two metrics that were used to measure the cost and effectiveness of the load relations mapping feature and that cover different areas of importance in simulator design: area, maximum path delay time, and runtime cost of memory initialization and finalization. Area is important because it is a limited resource. Many FPGAs are small in size and can be expensive to upgrade to larger sized models. It is important to keep additional area requirements to a minimum to permit this feature to be implemented on designs that already may be reaching maximum area limits on the FPGAs. The maximum path delay is used to determine how the feature affects the execution of other parts of the simulation. If the path delay increases the clock period it limits the speed at which the rest of the system runs and increases simulation time. Lastly the time required at runtime to perform memory initialization tasks is considered because it represents the amount of additional time in simulation independent of the feature's effect on the clock period. The tasks

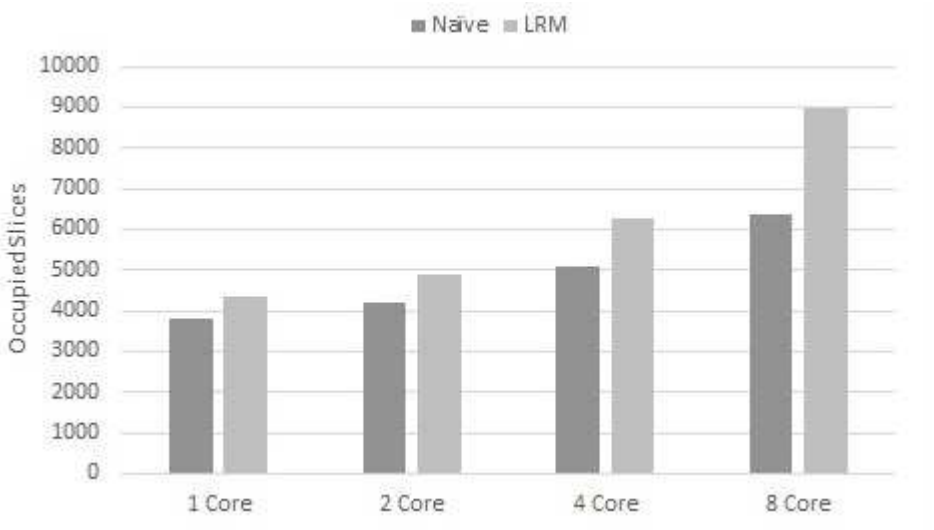

Figure 5.1: Area Comparison - PPCmp: Area comparison between simulator flagged to include feature and native simulator without feature.

of initialization and finalization are performed at the start and end of simulation and are a fixed overhead cost in the simulation runtime.

Figure 5.1 shows the area cost of implementing the load relations memory initialization feature. In the 1 core model the area is 8% larger on the feature enhanced version over the native simulator while at 16 cores it is a 31% increase in area size. On small core models little cost in area can be seen and it increases linearly with the number of cores. The growth rate is faster than that of the base cores and might be expensive to use in manycore models.

The feature has a very small effect on simulation speed based on the frequency at which the simulation can be run. As seen in figure 5.2 the maximum path delays are approximately the same for both the native simulator and the simulator with the feature included. These path delays are well below the minimum system period of 4ns are relatively consistent across all models showing that the feature does not increase the critical path and slow down the system clock.

While not fully measured it is important to mention the impact this feature has on total simulation run time. As state previously, the frequency at which the simulation can be run is not affected, causing the simulation to finished in the same amount of time with and without the feature. While the actual hardware simulation time is unaffected, the total time spent performing initialization tasks is increased with this feature. There is a fixed overhead incurred to perform the memory state initialization and finalization which adds on time proportional to the size of memory

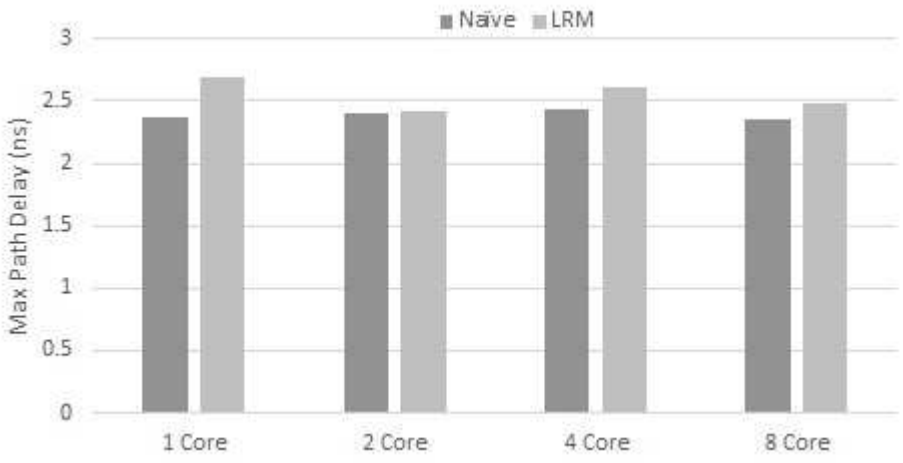

Figure 5.2: Timing Comparison - PPCmp: Timing comparison between simulator flagged to include feature and native simulator without feature.

being initialized. Even with large memories, this overhead is small when compared to the total simulation runtime and has a small effect on total runtime and simulation performance.

CHAPTER 6. CONCLUSION

Fast simulation is needed to permit the exploration and evaluation of new computer architectures. Co-simulation techniques show promise in their ability to improve simulator performance and will continue to be vital in advancing the design of new and better computer architectures. The ability to effectively design these simulators and to permit the use of a variety of data constructs is important to furthering this field of research.

This thesis has demonstrated a technique that permits the load of initial simulation state on the host to be mapped and initialized onto FPGAs without the need of explicit hand coding. It provides the ability to perform runtime address translation between host runtime memory addresses and profiling run addresses stored on the FPGAs. This permits runtime data, including stored addresses and pointers, to be transplanted between the host and FPGA without corrupting the simulation state.

This is accomplished by building upon previous object reachability techniques and creating a mapping of objects in memory and the pointers that relate them to each other. This load relations mapping is capable of discovering the runtime address of simulation state memories and provide the ability to find the simulation state initial values at runtime and to perform needed FPGA memory initialization of these memory values.

A practical implementation of the load relations mapping feature into the FAMEbuilder framework was discussed and evaluated showing the simplicity of adding this feature to a current co-simulation project. The cost of the implementation was discussed showing trade-offs in area, speed, and flexibility in implementing the feature.

Future work can extend these techniques to permit less structured dynamic memory regions, such as dynamic arrays of pointers to dynamic arrays, to also be supported in the load relations mapping to extend the range of addresses that can be translated and initialized at runtime.

The cost of implementing this feature can also be reduced by using more area efficient techniques to perform the memory initialization permitting a smaller footprint for the feature.

REFERENCES

[1] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A case for FAME: FPGA architecture model execution," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 290–301, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815999 1, 7

[2] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST methodology for high-speed SoC/computer simulation," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '07. Piscataway, NJ, USA: IEEE Press, 2007, pp. 295–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=1326073.1326133 1

[3] J. Emer, P. Ahuja, E. Borch, A. Klauser, and C. K. Luk. 1, 4, 6

[4] "IEEE standard for standard SystemC language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012. 4

[5] *IEEE Std 1666-2005: IEEE Standard SystemC Language Reference Manual*. IEEE, 2005. 4

[6] Z. Ruan, K. Rehme, and D. A. Penry, "SPRI: Simulator partitioning research infrastructure," in *2rd Workshop on Architectural Research Prototyping*, 2008, p. 2. 4, 8

[7] V. Koyyalagunta, H. Angepat, and D. Chiou, "HLS: High-level synthesis for highlevel simulation using FPGAs," 2010. 4

[8] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap scratchpads: Automatic memory and cache management for reconfigurable logic," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 25–28. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950421 4, 5

[9] 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 the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA '08. New York, NY, USA: ACM, 2008, pp. 77–86. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344684 4, 6

[10] A. Parashar, M. Adler, K. E. Fleming, M. Pellauer, and J. Emer, "LEAP: A virtual platform architecture for FPGAs," in *The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010. 4, 5

[11] H. So and R. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug 2006, pp. 1–6. 4

[12] X. Changqing, W. Mei, W. Nan, Z. Chunyuan, and H. So, "Extending BORPH for shared memory reconfigurable computers," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 563–566. 4

[13] Z. Ruan and D. Penry, "Partitioning and synthesis for hybrid architecture simulators," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 1859–1862. 4, 8

[14] U. Legedza and W. Weihl, "Reducing synchronization overhead in parallel simulation," in *Parallel and Distributed Simulation, 1996. Pads 96. Proceedings. Tenth Workshop on*, May 1996, pp. 86–95. 4

[15] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb 2011, pp. 406–417. 6

[16] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 15:1–15:32, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534925 6

[17] E. S. Chung, J. C. Hoe, and B. Falsafi, "ProtoFlex: Co-simulation for component-wise FPGA emulator development," in *Proceedings of the 2nd Annual*, 2006. 6

[18] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic, "RAMP gold: An FPGA-based architecture simulator for multiprocessors," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June 2010, pp. 463–468. 7

[19] J. R. Levine, *Linkers and Loaders*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. 10, 14

[20] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124 10

[21] IBM Corporation. (2015) Garbage collection roots. [Online]. Available: http://www-01.ibm.com/support/knowledgecenter/SS3KLZ/com.ibm.java.diagnostics.memory.analyzer.doc/gcroots.html 11

[22] T. J. McEntee, "Overview of garbage collection in symbolic computing," *SIGPLAN Lisp Pointers*, vol. 1, no. 3, pp. 8–16, Aug. 1987. [Online]. Available: http://doi.acm.org/10.1145/1317203.1317205 11

[23] J. F. Bartlett, “Compacting garbage collection with ambiguous roots,” *SIGPLAN Lisp Pointers*, vol. 1, no. 6, pp. 3–12, Apr. 1988. [Online]. Available: http://doi.acm.org/10.1145/1317224.1317225 11

[24] L. Presser and J. R. White, “Linkers and loaders,” *ACM Comput. Surv.*, vol. 4, no. 3, pp. 149–167, Sept. 1972. [Online]. Available: http://doi.acm.org/10.1145/356603.356605 14

[25] J. M. Hart, *Windows System Programming (3rd Edition)*. Addison-Wesley Professional, 2004. 14