

BINARY SYNTHESIS WITH MULTIPLE MEMORY BANKS TARGETING ARRAY REFERENCES

Yosi Ben Asher

Computer Science Dep.
Haifa University
email: yosi@cs.haifa.ac.il

Nadav Rotem

Computer Science Dep.
Haifa University
email: rotemn@cs.haifa.ac.il

ABSTRACT

High-Level Synthesis (HLS) is the field of transforming a high-level programming language, such as C, into a register transfer level(RTL) description of the design. In HLS, Binary Synthesis[14] is a method for synthesizing existing compiled applications for which the source code is not available. One of the advantages of FPGAs over software is the availability of multiple memory banks. Until now, binary synthesis systems have not made use of the multiple memory banks on FPGAs. In our work, we decompile the binary executable into an intermediate representation, and we target architectures with multiple memory banks and multiple memory ports. We present methods for detecting memory regions and synthesis of the decompiled code. The proposed methods accelerate the execution time of applications which use multiple memory regions concurrently.

1. INTRODUCTION

Accelerating software execution times is an important and well-researched field both in the academic world and in the industry. There are numerous fields of research and techniques to accelerate and optimize execution time. Advanced compiler techniques to optimize, vectorize, and parallelize code are used by compilers to increase application performance. Virtual Machines, such as Java JVM[2], implement just-in-time [10] compilers to provide the best performance to programs which are encoded in bytecode. Binary translation systems translate compiled binaries from one architecture into another. The binary is first emulated one instruction at a time. When a frequent path in the emulated binary is detected, the binary translation system creates an efficient representation of the path which is compiled into native machine code. Field Programmable Gate-Arrays(FPGA) are often used to accelerate programs. The FPGA is a popular platform for hardware acceleration. FPGAs are reconfigurable at run-time and at the same time enjoy the properties of hardware circuits.

Programming FPGAs in languages such as Verilog and VHDL is considered more difficult and time-consuming than

programming high-level software languages. Efficiently compiling from high-level languages, such as C or Java, to gate-level may reduce time-to-market, ease verification, and lower design costs. High-level synthesis tools must achieve satisfactory performance in terms of design size, execution time, and power consumption. These restrictions are the fundamental problems in HLS[4]. Unlike software compilers, for which the architecture of the processors is known, in HLS, a hardware scheduler is responsible for creating the architecture for the program it compiles. The compiler is free to assign as many registers, memory ports, functional units, and buses as necessary as long as the result is synthesizable to a chip.

Binary synthesis[14] takes a different approach than the aforementioned tools. It is a technique for performing high-level synthesis on precompiled applications. The synthesis system takes an existing compiled application and accelerates it using an FPGA. The "hot" loops in the software are selected and translated into hardware circuits. The rest of the application remains unmodified.

In binary synthesis there are several possible configurations for acceleration. One approach is to use an extension card with an FPGA. The acceleration card with the FPGA is added to a standard computer which uses it to access the FPGA. Another method requires an embedded system in which the CPU is a soft-core (such as Xilinx MicroBlaze) which is synthesized on an FPGA. The software is executed on the soft-core while the accelerated code is synthesized into the FPGA fabric. Other configurations, such as System-on-chip designs, where the CPU and FPGA are on the same chip (such as Virtex 4 by Xilinx), are also an ideal platform for binary synthesis.

Binary synthesis systems operate on binary applications for which the source code is not available. Binary synthesis systems disassemble[8] and reconstruct the compiled binary in several stages. First, the Control Flow Graph (CFG) is restored, after which the Data Dependency Graphs (DDG) are reconstructed. Once the CFG and DDG exist, it is possible to emit C code which represents the original executable or parts of it. Once the original executable is represented in C,

it can be recompiled into a hardware description language, much like the aforementioned HLS tools. Most of the code in the original executable or shared object remains unmodified and only select parts of the code are changed. Only the parts which are selected for synthesis are translated into hardware and modified in the original executable. The modified code is patched with a "branch" instruction that points to a stub function which starts the execution of the acceleration hardware. The patching of the original binary is called Binary Updating.

Vahid and Stitt[14] demonstrate that synthesis of software binaries into hardware circuits can be beneficial in several situations. One disadvantage of binary synthesis compared to the HLS synthesis approach is that some information is lost in the process of compilation to the binary file. They note that without the array detection, memory structures cannot be synthesized efficiently. In another study, Vahid, Najjar et al., show that through using decompilation techniques[13] that reconstruct arrays and loop structures, they are able to reconstruct the same memory architecture as the original binary applications. They use "Smart Buffers"[6] as a mechanism to hide memory fetch latency from the global host memory. By recovering arrays, they allow the ROCCC[7] synthesis system to use Smart Buffers and accelerate the generated circuit.

Typically, on an FPGA, there are multiple memory units (Block-RAMs). Each of these memory units have multiple access ports (usually only two). The FPGA memory architecture differs from that of a CPU which has only one central memory bank with only one memory port. We note that some microprocessors, especially DSPs, have different memory ports for instructions and data. However, usually only one bank is available. One of the major advantages of FPGAs over CPUs is that the access speed in terms of bandwidth, port number and latency is much greater.

Many programs use loops which access several memory locations. For example, a loop which copies a region of memory uses two arrays: the source array and the target array. When these loops are compiled into assembly code, the compiler places the arrays in a single memory bank. This behavior of the compiler is optimal for CPUs but not for FPGAs. If we could place the different arrays in multiple memory banks, then we could benefit from the concurrent memory access. In the example below, we see a sample code for multiplying values from two different arrays. It is obvious that placing array A , B and C in three different memory banks on an FPGA would result in a much better design. A good synthesis system would implement a design which would read values from A and B , perform multiplication, and store the results of the previous iteration to C in each clock cycle. Without multiple memory banks, the synthesis system would have to create a design which would wait for the different values to be loaded and stored to the arrays.

This would result in a circuit which takes 3 cycles for each iteration instead of one.

```
for (int i=0; i<n; ++i) {
    A[i]=B[i]*C[i];
}
```

Another case which would obviously benefit from multiple memory banks is the following:

```
for (int i=0; i<n; ++i) {
    B[i]=T[A[i]];
}
```

If table T , in the code above, fits on an FPGA, then the application could benefit from placing array T on a different memory bank. The detection of table T poses even more problems on the detection of arrays, since it is referenced in arbitrary locations, much like a struct or a linked list.

Note that the mapping between arrays and memory banks is not trivial. For example, in the following code only two memory banks should be used (A and C).

```
for (int i=0; i<n; ++i) {
    A[i]=A[i-1]*C[i]+C[i-2];
}
```

In our work, we have implemented a binary synthesis system which synthesizes compiled shared objects into the Verilog hardware description language. The synthesis system uses SystemRacer[1] as the synthesis engine. In our work, we present two methods for detecting arrays and tables and propose two architectures for implementing binary synthesis memory management. After detecting the memory regions used by each memory operation in the original binary, we may assign memory banks to each unique region and thus enjoy the benefits of multiple memory banks.

2. PROPOSED TECHNIQUE

Vahid, Najjar et al., describe the use of Smart Buffers[13] in binary synthesis. In their work, they implement one memory bank which fetches data from the host. We refer to this memory region as "global memory". The global memory is a memory unit which is responsible for the entire address space of the program. The global memory module fetches memory regions from the host computer memory and may implement caching of data. In our work, we propose extending the global memory by adding additional memory banks which are local to the accelerator. These memory banks are synthesized especially for arrays which are detected in the program. By dedicating memory banks to frequently used arrays, we are able to accelerate the execution of programs.

In order to assign memory banks to arrays and tables, we must first detect them properly. The first method for detecting arrays and tables is the dynamic method. In the dynamic



Fig. 1. Memory ranges of opcodes from the memory profile

method, we use a memory profiler to obtain run-time information about the memory addresses in use. At run-time, it is possible to create an address range for each load/store instruction. We can create a memory bound estimation for each opcode which accesses memory. Each opcode in the binary application has an estimated range of addresses.

When the application is decompiled, each load/store opcode which has a memory bound estimation is compared to other opcodes and they are marked as one group. We discuss the grouping conditions below. Each group defines a memory region and is then assigned to a different memory bank. Groups which reference memory ranges which could not be assigned to a specific memory bank are assigned to the global memory bank. The dynamic method will be able to detect arrays with sequential access in addition to tables which are accessed randomly.

After the memory profiling is completed, there is a list of opcodes which performed memory operations. For each opcode, there is a range of memory addresses that were accessed. Figure 1 shows two possible memory maps. On the right, we see a well-behaved memory map. Several opcodes access three distinct memory ranges.

In the case of the well-behaved memory maps, it is easy to partition the memory map into multiple banks. By grouping together all of the opcodes which intersect, we are able to detect groups of opcodes which access a single memory region. We can discern a memory bank from the global memory by detecting memory regions which do not intersect with others.

The second proposed method for detecting arrays is static analysis of data structures. In this method, we detect memory regions with sequential access patterns. By detecting the memory regions which are accessed sequentially, we are able to find good candidates for separate memory modules. In order to detect a sequential memory access, we use the IVR[5] compiler analysis. This analysis locates induction variables in a loop. After we identify the induction variable, detecting array access is simple. A sequential array access is a memory operation on a pointer. The pointer needs to hold an address which is calculated using the following pattern $Address = IV * S + B$. Where IV is the induction variable, S is typically the size of the array element type and B is the base address of the array. Both B and S must be constant inside the loop. The offset B can be a variable or a function parameter. Two memory references $Address_i = IV_i * S_i + B_i, Address_j = IV_j * S_j + B_j$ are grouped to the

same bank if $B_i \approx B_j$ and $S_i = S_j$. Thus, it is likely that memory references resulting from $A[i], A[i - 1], A[i + 8]$ will be allocated to the same memory bank.

In the architecture we propose, the hardware circuit aborts the execution of the accelerated method. In the case of a detected range violation, the host computer will execute the original software implementation of the code. When an invalid memory address is requested from a memory bank, an interrupt is fired and the host computer aborts the execution of the circuit. By allowing the hardware circuit to fail, we allow the binary synthesis system to synthesize code which is likely but not guaranteed to succeed. This solution is especially suitable for cases where the pointer addresses are known only at runtime and there is a chance of successful mapping of memory banks.

3. IMPLEMENTATION

Our system is based on the LLVM [9] compiler infrastructure. First, we disassemble the binary and reconstruct compiler data structures. Next, we apply standard compiler optimization passes to optimize the code and remove the instruction-set overhead. We then detect the static arrays and use the dynamic run-time information to detect other memory regions. Finally, we synthesize the disassembled code into Verilog.

In our research, we developed a dynamic profiler for detecting memory structures in compiled applications. We have implemented a program for tracing and recording memory access patterns in running processes. We execute the target application using the tracer program which executes the target application in "single-step" mode. After each instruction, we record the information needed for determining which memory addresses are accessed. We record the 16-bytes of memory following the instruction pointer as well as the state of all of the hardware registers. We used the diStorm[3] disassembly library to decode the opcodes into an assembly representation.

We note that profiling applications dynamically in a single-step mode is slow. It is possible to speed up the process by tracing only interesting code segments such as certain libraries or methods. This can be achieved by carefully placing break points at the entrance and exit points of these segments. Another approach would be to implement a kernel-space debugger.

4. EXPERIMENTAL EVALUATION

To evaluate the proposed techniques, we collected several kernels from different sources (discussed below). The kernels were extracted from the following programs: Cft1st (Continuous Fourier transform) is a part of the general purpose FFT package by OOURA. Cplextrn is a complex transpose used in the FFTW package. Crc32 is the Cyclic Re-

dundancy Check used in many network protocol implementations. Fir is the finite impulse response filter. Histogram is a program which calculates 256-bin histogram of integers. Conv is a reference row convolution filter. Both Histogram and Conv are from the NVidia CUDA SDK[12]. Popcnt is a program which counts the number of 1's in an array using 8-bit lookup table. RC4 is the widely-used stream cipher. Wavelet is a discrete wavelet transform. Yuv2rgb is a color conversion program from LIBOIL[11]. Liv.kernel 5, 10 and 14 are from the Livermore Loops Benchmark.

We used our binary synthesis system to disassemble binary files which were compiled using gcc-4.1 with limited optimizations (-O1). The recovered binaries were compiled as a shared library for the x86 architecture and were stripped of any debug symbols.

We tested both the static method and the dynamic method for finding arrays and tables. When using the static method, all of the sequential arrays were detected. In the case of Popcnt and Crc32, the static array detection pass did not detect the look-up tables. When using the dynamic method, the tracer program was able to detect all of the memory regions. The dynamic method did detect the look-up tables which were not detected with the static method.

In the experiments, we compared test programs that used a single memory bank to programs which used multiple memory banks. Each one of the banks had two memory ports. In most of the tests we ran, we witnessed a major improvement in performance when multiple banks were used. The number of arrays used by the program is correlated with the performance increase. In most of the tests, the synthesis was able to make use of multiple memory operations concurrently. In other words, in each clock cycle, several memory banks were accessed. Additionally, accessing the memory was the bottleneck of most of the applications.

5. CONCLUSIONS

Previous works have shown that binary synthesis of pre-compiled applications may accelerate the execution time of programs. Understanding the program's memory usage patterns and data structures is critical in synthesizing an efficient accelerator. Until now, binary synthesis systems have not made use of the multiple memory banks on FPGAs. The main contribution of this study stems from its bridging the use of multiple memory banks for hardware applications and the extraction of memory structures from executables. The combination of these two techniques allows for improved performance. In this work, we have proposed two methods for recovering memory data structures from the compiled applications. We have demonstrated, in our experiments, that applications which use multiple memory banks benefit greatly.

6. REFERENCES

- [1] Yosi Ben-Asher and Nadav Rotem. Synthesis for variable pipelined function units. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–4. IEEE Computer Society, 2008.
- [2] Michał Cierniak and Wei Li. Just-in-time optimizations for high-performance Java programs. *Concurrency: Practice and Experience*, 9(11):1063–1073, 1997.
- [3] Gil Dabah. distorm, the disassembly library, 2005. <http://ragestorm.net/distorm/>.
- [4] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.
- [5] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems*, 17:85–122, 1995.
- [6] Zhi Guo, Betül Buyukkurt, and Walid Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 249–256. ACM Press, 2004.
- [7] Zhi Guo, Betül Buyukkurt, and Walid Najjar. Optimized generation of data-path from c codes for fpgas. In *Int. ACM/IEEE Design, Automation and Test in Europe Conference (DATE 2005)*, pages 112–117. IEEE Computer Society, 2005.
- [8] C. Kruegel, W. Robertson, F. Vaur, and G. Vigna. Static disassembly of obfuscated binaries, 2004.
- [9] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
- [10] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In *Programs as Data Objects, Second Symposium, PADO 2001*.
- [11] David Schlee. Library for optimized inner loops. <http://liboil.freedesktop.org/wiki>.
- [12] NVidia CUDA SDK. http://www.nvidia.com/object/cuda_showcase.html.
- [13] Greg Stitt, Zhi Guo, Frank Vahid, and Walid Najjar. Techniques for synthesizing binaries to an advanced register/memory structure. In *In FPGA 05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 118–124. ACM Press, 2005.
- [14] Greg Stitt and Frank Vahid. Binary synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–30, 2007.