Communicating Process Architectures 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

The Trebuchet

John Campbell
Northrop Grumman Corporation, 888 South 2000 East, Clearfield, Ut 84015-6216
john.campbell@ngc.com

G. S. Stiles
Computing Laboratory, University of Kent, Canterbury CT2 7NF
dyke.stiles@ece.usu.edu

Abstract. The Trebuchet is a hardware architecture for machines implemented from conventional software source programs. It is pseudo-asynchronous in that it decouples the system clock from the computational logic, reducing electromagnetic interference, smoothing current draw, and reducing pipeline latency, key benefits of asynchronous designs. Performance of example programs is given.

1. Introduction

This paper discusses the design of the Trebuchet, a pseudo-asynchronous micropipeline, and proposes a new architecture for high performance reconfigurable computing. It grows out of an earlier Trebuchet effort we called SMAL [1].

This version of the Trebuchet consists of a linear pipeline of configurable computing elements. Each element (composed of one or more FPGAs (Field Programmable Gate arrays)) executes in succession and is then reconfigured. The number of required computing elements depends on the length of time required to finish one "step" and the time required to reconfigure an element. The system is composed of many interacting state machines (Fig. 1.) Each state machine, while entirely synchronous, behaves asynchronously in the sense that stage latency depends on the complexity of the computation performed rather than on the clock period. The clock period, rather than being dependent on the longest logic chain, is set by the time required to exchange handshaking signals with immediate neighbors [2].

On the global scale, the Trebuchet resembles Sutherland's asynchronous Micropipelines [3], inheriting many of their behavioral characteristics, despite being completely synchronous in the details. Hence we describe the Trebuchet as Pseudo-Asynchronous. We claim that pseudo-asynchronism allows the flexibility required to implement software as a cohesive hardware machine, and that the benefits normally ascribed to asynchronous machinery [4] may be achievable. Specifically, throughput

should depend, as in conventional asynchronous circuitry, on the average stage latency rather than the longest logic chain. Likewise, current draw should be smoothed with attendant reductions in radiated EMI (Electromagnetic Interference).

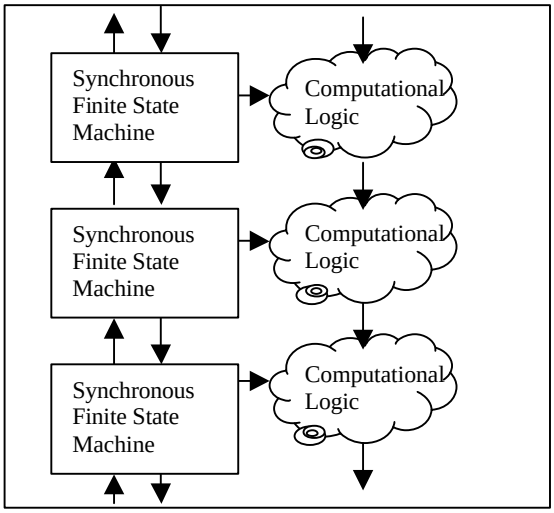

Fig. 1. A Trebuchet pipeline consisting of FSM-controlled stages.

The thrust of our research has been to develop a methodology for seamless implementation of hardware and software functionality. It has applications in both the fields of embedded systems and reconfigurable computing engines. The first (embedded systems) is concerned with dividing functionality between software and hardware (generally application specific integrated circuits (ASICs)), whereas the latter seeks to off-load time-critical functions to temporary circuits configured into FPGAs.

In contrast to conventional projects where software is prepared in an environment of existing and stable hardware, embedded computer systems typically require parallel development of hardware and software components [5]. Because the respective disciplines of hardware and software development are commonly conceived of as quite different, early decisions about task allocation are made which have profound and irreversible consequences on the ultimate cost and performance of the system. Consequently, much of the research in embedded system technology is devoted to blending the development methodologies by deriving both hardware and software from high-level descriptions so that decisions can be delayed as long as possible and are demonstrably correct when made [6].

Because conventional hardware design is tedious and foreign to most software practitioners, generating hardware from software has been an important research goal [6]. Generally speaking, these approaches target reconfigurable machinery hosted on FPGAs. However, reconfigurable co-processors tend to execute their functions rapidly and then remain idle for long periods of time. Consequently, the question of how to dynamically reconfigure the co-processor has also become important [7], [8].

Our overall goal has been to develop a methodology in which the use of CPUs, reconfigurable logic implemented in FPGAs, and permanently configured logic in an ASIC are all parameters in a scheduling problem. But because available chip real estate

allows implementation of only small portions of a program in hardware, it seems desirable to develop an approach that deals with multiple chips and the consequences of chip and board boundaries.

We seek a methodology that automatically allocates portions of a program to a network of execution resources based on single-threaded software execution profiles. A methodology that constructs systems using both hardware and software can best take advantage of the available execution resources. Code written in a high level language (e.g. Java) may execute as machine instructions or directly as hardware. The decision should depend on the needs of the work to be done and the resources available to do it.

In [9] we presented a theory of hierarchical network domains and applied it to systolic networks being optimized for maximal throughput. This research expands that work to address virtual networks of reconfigurable components, allowing a uniform treatment of devices with quite different reconfiguration characteristics (i.e. dynamically and statically reconfigurable FPGAs, ASICs, and CPUs).

We chose Java as the application language because the Java Virtual Machine (JVM) has a simple and regular addressing scheme without registers [10], and because the interpreter makes it easy to gather execution statistics that may be used in mapping experiments. Our research [11] indicated that Java execution is predictable enough that transformation of portions of an application to hardware is possible.

Fleischmann and Buchenrieder [12] are also using Java to study reconfigurable hardware systems, but do not generate hardware automatically, as our system does. Hutchings et al. [13] are doing low-level hardware design with Java-based JHDL, but their tool is not aimed at high performance pipeline systems. Handel-C is a similar system based on the C language; this approach associates assignment operations with latches [14], whereas we associate assignments with wires and latch much larger aggregates.

2. The Trebuchet System

The Trebuchet runs hardware compiled from Java source code. We modified Jikes [15], an open source java compiler originally from IBM, to include extra information we needed for the conversion to hardware. The output of Jikes is a standard Java class file. We obtain profile information from a modified JVM (Java_g, part of the Sun Java JDK). The profile also includes segmentation of the Java byte-codes into basic blocks and descriptors for the structure of *for* loops, *if* statements, etc.

There is considerable opportunity for fine-grained parallelism. While parallelism is, in principal, possible to detect automatically, we added the keyword *par* to the syntax parsed by Jikes. *Par* signifies to the VHDL translator that a *for* loop is vectorizable.

The Java byte-codes are translated to VHDL by analyzing the basic block contents. Stack and Memory references become accesses to wires (thus being essentially compiled out) and successive op-codes become, for the most part, cascaded blocks of combinational logic. Array accesses become accesses to RAM. Since each array resides in its own RAM, concurrent access to different arrays is supported. Concurrent access to the same array must be arbitrated across the program.

We targeted our hardware at the Xilinx V800 FPGA. With capacity of 800,000 gates, there is room for moderate sized software experiments. In the future we will address designs involving multiple chips so that programs of arbitrary size may be executed. Hardware configuration files are generated by standard Xilinx tools [16].

2.1 Modifications to the Jikes Java Compiler

The Java code is compiled by a modified version of the Java compiler Jikes. We added keywords to the otherwise standard Java syntax recognized by Jikes. As noted above, in principle the compiler could have been modified to recognize parallelizable and systolizable loops [17]. At some point in the future, we intend to do this.

In addition to the *par*, we also included the keywords *netstart*, *netend*, and *expose*. *Netstart* and *netend* indicate respectively the beginning and end of the code to be analyzed for hardware mapping. *Expose* designates variables that are required as output from the execution engine. All of these are expediencies that could, in principal, be automatically recognized by a compiler.

The structures which may be inserted into the Java class files include *start* tags, *stop* tags, and *parallel loop descriptor* tags. The latter designate *for* loops for vectorization. The byte code interpreter has special code added to it to detect this extra information in the execution stream of the program. Because it is desirable that code thusly modified also be executable by conventional JVM platforms, the tags are structured so that a conventional JVM will simply jump around them.

We needed to format the tags so that they could be unambiguously recognized in the JVM execution stream. Since the compiler never generates a jump-to-self instruction, we can use this instruction as our tag: these are constructed as a jump followed by a jump-to-self, followed by tag specific information. Fig. 2 illustrates this format.

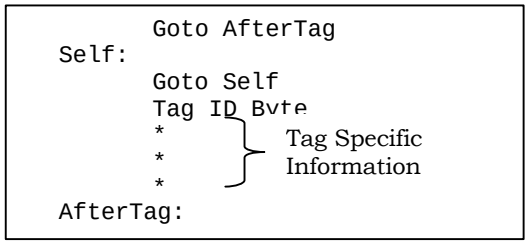

Fig. 2. Format of description tags embedded in Java execution stream.

2.2 Modifications to the Java Virtual Machine

One of the purposes of the Trebuchet is to experiment with mapping of regions of software onto hardware regions. The theory of this mapping [9] depends on profile information on the number of times communication arcs are utilized, rather than the number of times nodes are visited – a standard profile. Consequently the JVM was altered to collect *transfer of control* statistics. Java_g was tailored to output a file of

bytecodes segmented into basic blocks (basic blocks are sequences of code which terminate at program jumps).

2.3 Bytecode Translation to VHDL

Trebuchet, written in Common Lisp, translates the basic block and profile information provided by the modified JVM. Trebuchet symbolically traverses each basic block, generating combinational logic corresponding to the sequence of instructions. Some instructions (e.g. multiplication) are impractical to configure as purely combinational logic and necessitate further segmentation. Trebuchet also constructs hierarchical components, such as *for* loops, that consist of a controlling FSM (finite state machine) and other subcomponents.

A basic block is a sequence of code unbroken by changes in sequential flow (except at the terminus). Thus there are not multiple paths of execution within a basic block. Trebuchet traverses a basic block, examining each instruction. Bytecodes that manipulate memory (either stack or variable store) such as IPUSH rearrange the set of working wires. Operations that produce values (e.g. IADD) take their inputs from the set of working wires (deleting them from the set) and introduce new wires with the outputs. Trebuchet translates each basic block to a combinational net of logic. Fig. 3 shows an example basic block in source code and as hardware logic.

Many compilers manipulate stacks internally and map stack locations to registers. Pushing an object on the stack, while conceptually moving the entire stack, in reality only changes the association between register names and stack offsets. Trebuchet does this with the set of wires representing program memory and the stack.

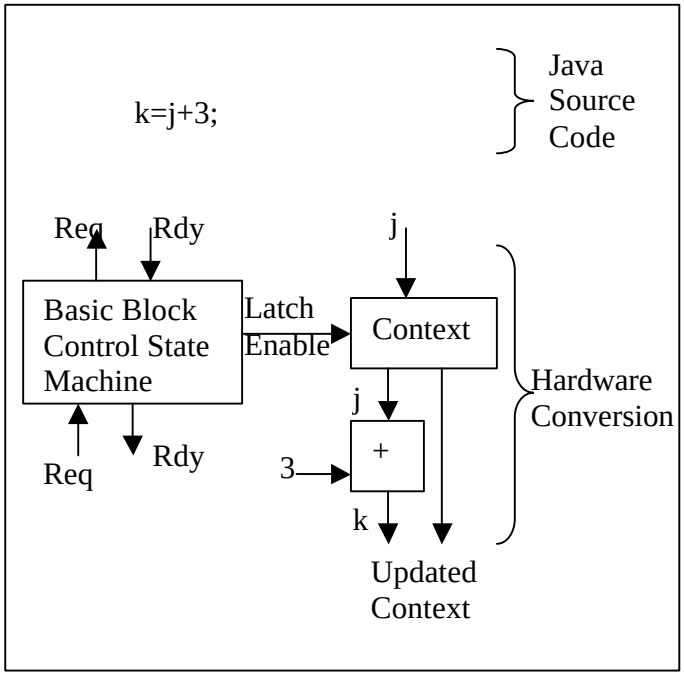

Fig. 3. Example translation of a fragment of Java code.

2.4 Vectorized Loops

Because the algorithms we want to investigate with Trebuchet manipulate programs that stream data through operators, we needed a way to generate code that could execute systolicly. Java does not have a parallel operator, so we added one.

Conceptually, our *par for* loops have four parts. These include the *initialization_clause, the end_test,* and the *step_clause* that conventional Java and C share. The *loop_body* is run overlapped. As Fig. 4 shows, a *par for* loop is organized with a tight loop that repetitively steps the loop variable, tests the termination condition, and initiates an entry into the pipeline. All three of these operations are executed in parallel to minimize the latency between subsequent pipeline entries.

Pipeline initiation may be thought of as a thread that executes a particular loop iteration. This construction is not far removed from loop vectorization, a well-studied topic in computer science [17] and could, in principle, have been accomplished automatically. But for our purposes, it is enough to use the *par* keyword to designate code that can be validly overlapped in execution.

Trebuchet generates a vectorized loop from the body of the Java *for* loop and the control clauses specified with it. It rearranges the controls and forms a pipeline from the succession of basic blocks in the body of the loop. The test condition, the stepping of loop variables, and the first stage of the pipe are all initiated in parallel.

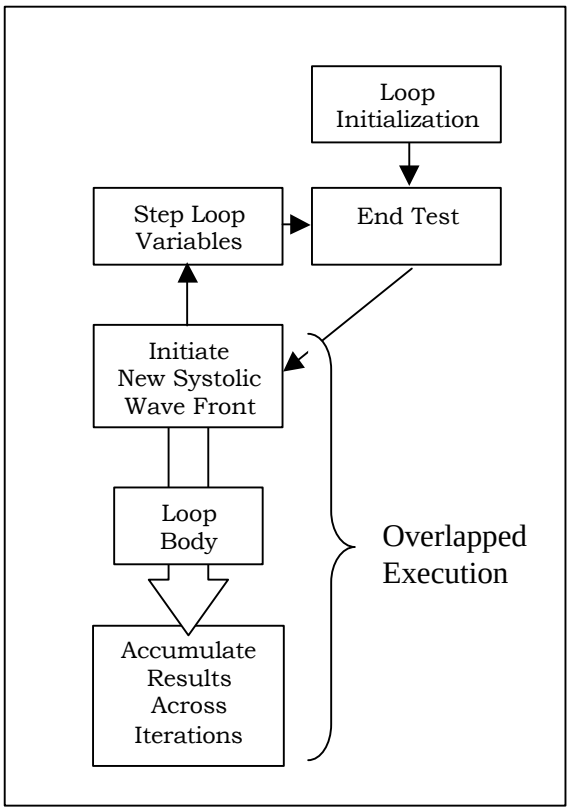

Fig. 4. Vectorized for loop.

Each iteration depends on the 'current' loop variables, as does the end test. Since these are all executed in parallel, the loop is unrolled to precompute the data. Additionally, each iteration carries forward a flag that signals the final iteration of the loop. The loop test, in Java, is intended as a condition for breaking out of the loop, and thus signals on *completion* of the final iteration.

The end condition must be propagated down the pipe because of the behavior of the last pipeline stage. It consumes each thread until signaled that the last iteration has arrived. In this special case, it handshakes its results out to whatever follows the loop.

Since each iteration propagates an end test value corresponding to the next iteration, and the test itself depends on stepped variables, each cycle must precompute index variables that are *two* iterations ahead and a test value that is *one* iteration ahead. The control loop is unrolled to obtain these phase relationships. One consequence of this is that, like old style FORTRAN *do* loops, *par for* loops must be guaranteed to execute once. Another restriction on valid *par for* loops is that step and test clauses not have side effects or access arrays.

2.5 Conditional Code

The Java compiler handles conditionally executed code by jumping around it. In vectorized code, successive executions (threads) must not be allowed to overtake and pass prior threads. Consequently, Trebuchet threads propagate from stage to stage, even where execution is suppressed. The boolean test result propagates through the range of the conditional execution. At each stage, it suppresses update of the values passed to the next stage by switching a multiplexer. Fig. 5 illustrates this operation. Similar measures have been taken by makers of vector processing units for conventional computers [17]. Note that even though pipeline threads must traverse stages for which execution is suppressed, such traversal is rapid because the controller FSM immediately transitions to output states without waiting for the logic to propagate through the computational logic.

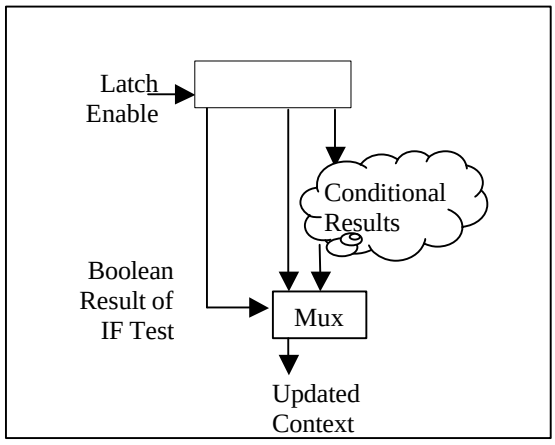

Fig. 5. Conditionally utilized results are switched with a multiplexer.

2.6 Array References

Java arrays are dynamic in the sense that they may be created at any time, moved around as needed to optimize garbage collection, and reclaimed by the garbage collector when abandoned. We did not want to subject the hardware generated by Trebuchet to the performance penalties inherent in such manipulation, so we chose to map array creations to static arrays created in the FPGA at configuration time.

If, during the course of symbolic modeling of stack and variable store, a reference is made to a location identified with a particular array, the mechanism to access that array is constructed. If the array reference cannot be ascertained, the module is marked as not being compatible with realization as hardware, and would necessarily be limited to bytecode execution by the JVM.

3. Pseudo-Asynchronous Execution

Trebuchet, when generating VHDL, calculates the length of the logic path for each pipeline stage and generates a FSM controller with enough wait states to allow signals to propagate. In order to decouple the clock period from this path, Trebuchet generates multi-cycle clock specifications for the Xilinx tools [18]. This allows the clock period to be driven by the exchange of handshaking signals rather than by the critical path through the combinational logic. This means that for lightly loaded conditions, the average stage delay dominates the pipe transit time, instead of the worst-case stage delay. This is one of the advantages touted for the asynchronous Micropipeline [4].

Another desirable trait of asynchronous circuitry is that, without a synchronizing clock, logic transitions are very well distributed in time. This minimizes current draw on the power supply and reduces the level of radiated EMI (electromagnetic interference) [4]. The tendency of synchronous logic is to have a well-defined signature of successive gates transitioning (and drawing power). It is expected that heavy usage of multi-cycle logic paths will have the effect of smearing these signatures, thus obtaining some of the advantage of purely asynchronous circuitry.

Drawbacks of asynchronous circuits include sensitivity to signal noise, performance dependent on temperature and process variations, and incompatibility with conventional FPGA tools [4]. The Trebuchet avoids these difficulties by being, at heart, synchronous. It combines the best aspects of both worlds.

4.0 Trebuchet Performance

The example programs [19] are derived from the well-known Hail-Stone Algorithm. The Hail-Stone applies the following algorithm:

Given N_i,
If N_i is even, $N_{i+1} = ½ N_i$
If N_i is odd, $N_{i+1} = 3N_i + 1$

The name “Hail Stone” derives from its characteristic of unpredictably increasing and decreasing in magnitude, the way a meteorological hailstone repeatedly rises and falls

during its formation. The test program given in Fig. 6 computes the first 10 iterations when N_0 is set to 37.

The parallel bulk hailstone program is given in Fig. 7. It utilizes the nonstandard keyword *par* to direct the vectorized execution of the *for* loop, thus causing each of the computational steps to become stages in a 10-step pipeline. The loop limit was increased to 40 to keep the pipeline full longer during its execution. Test results for Simple_hail and Bulk_hail_par are shown in Table 1. There is an increase of almost 19-fold in throughput associated with the parallel version, attributable to pipelining effects.

5.0 Conclusions

The Trebuchet is an on-going effort. The project has already provided insight into the nature of hardware/software systems, and promises to provide some of the benefits commonly associated with asynchronous hardware design. Executing as hardware, code written in a traditional software language offers to finally join the disparate fields of hardware and software system design. The medieval Trebuchet was a marvelous example of technology used to loft heavy stones and hurl them at enemy fortresses. Our Trebuchet is, we hope, a tool for assaulting the bastions of complex system design.

```
public class simple_hail {
      public static void main(String args[]){
                  netstart;
                        int i,x;
                        x=37;
                        for(i=0;i<10;i++){
                           if(((x>>1)<<1)==x) x=x>>1;
                           else x+=x+x+1;
                        }
                        expose {x=x+1;}
                        netend;
                        System.out.println("\n hail(37,10):"+x);
            }
}
```

Fig. 6. Simple_hail.java source code.

Table 1. Test Results

	Simple_Hail	**Bulk_Hail_Par**
JVM Instructions	424	8,044
Equivalent Gates	18590	76,332
Fraction of XCV800 FPGA Utilized	9%	41%
Clock Frequency	36 Mhz	39 Mhz
Execution Time	11.36 µs	15.8 µs
Equivalent JVM MIPS	27	509

```
public class bulk_hail_par {
    public static void main(String args[]){
        netstart;
        int i,x;
        x=0;
        par for(i=0;i<40;i++){
            x=i;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
            if(((x>>1)<<1)==x) x=x>>1;
            else x+=x+x+1;
        }
        expose {x=x+1;}
        netend;
        System.out.println("\n "+x);
    }
}
```

Fig. 7. Source code for bulk_hail_par.java.

References

[1] J. Campbell, "Experience with a Reconfigurable Java Machine," *Int'l. Conf. on Parallel and Distributed Processing Techniques and Applications*, pp. 2459-66, June 26-29, 2000.

[2] J. Campbell and G. S. Stiles, *The Trebuchet, a Pseudo-Asynchronous Micropipeline Execution Engine*, International Conference on Systems Engineering 2002, University of Nevada, Las Vegas, Aug 2002.

[3] I. Sutherland, "Micropipelines," *Turing Lecture*, *Communications of the ACM*, vol. 32, pp. 720-738, June 1989.

[4] A. Davis and S. M. Nowick, "An Introduction to Asynchronous Circuit Design," *University of Utah Technical Report UUCS-97-013*, Sep. 1997.

[5] P. M. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11-18, Mar. 1993.

[6] W. H. Mangione-Smith, "Seeking Solutions in Configurable Computing," *IEEE Computer*, vol. 30, no. 12, pp. 38-43, Dec. 1997.

[7] B. Hutchings and M. J. Wirthlin, "A Dynamic Instruction Set Computer," *Proc. the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99-107, Apr. 1995.

[8] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Through Run-Time Constant Propagation," *Field Programable Gate Array Workshop*, pp. 86-92, Feb. 1997.

[9] J. Campbell and B. Abbott, "Gear Train Theory: An Approach to the Assignment Problem Providing Tractable Solutions with Measured Optimality," *Int'l. Conf. on Parallel and Distributed Processing Techniques and Applications*, vol. II, pp. 986-95, June 30-July 3, 1997.

[10] J. Meyer and T. Downing, *Java Virtual Machine*, O'Reilly, 1997.

[11] P. Narayanaswamy, "Dynamic Arithmetic-Logic Unit Cache," *MS Thesis, Department of Electrical Engineering*, Utah State University, 1999.

[12] J. Fleischmann and K. Buchenrieder, "Prototyping Networked Embedded Systems," *Computer*, vol. 32, no. 2, pp. 116-9, Feb. 1999.

[13] http://www.jhdl.com/release-latest/docs/overview/intro.html.

[14] S. M. Loo, B. Earl Wells, N. Freije, and J. Kulick "Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time Systems *Proceedings of the Southeastern Symposium on System Theory (SSST-2002)*, pp. 6-10, Huntsville, AL, March 18-19, 2002.

[15] http://oss.software.ibm.com/developerworks/opensource/jikes/.

[16] *Development System Reference Guide*, Xilinx, Inc., 2001.

[17] J. L. Hennessy and D. A. Patterson, "Computer Architecture a Quantitative Approach," *Morgan Kaufmann Publishers, Inc.*, pp. 371-380, 1990.

[18] *Constraints Guide*, Xilinx, Inc., 2001.

[19] J. Campbell, "Efficient Automatic Mapping of Software to Hardware," Ph.D. Dissertation, Utah State University, 2002