Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor

Trevor Meyerowitz,
Alberto Sangiovanni-Vincentelli
University of California at Berkeley
Berkeley, California, USA
{tcm, alberto}@eecs.berkeley.edu

Mirko Sauermann, Dominik Langen
Infineon Technologies
Munich, Germany
{Mirko.Sauermann,
Dominik.Langen}@infineon.com

ABSTRACT

A generic and retargetable tool flow is presented that enables the export of timing data from software running on a cycle-accurate Virtual Prototype (VP) to a concurrent functional simulator. First, an annotation framework takes information gathered from running an application on the VP and automatically annotates the line-level delays back to the original source code. Then, a SystemC-based timed functional simulator runs the annotated source code much faster than the VP while preserving timing accuracy. This simulator is API-compatible with the multiprocessor's operating system. Therefore, it can compile and run unmodified applications on the host PC. This flow has been implemented for MuSIC(Multiple SIMD Cores) [6], a heterogeneous multiprocessor developed at Infineon to support Software Defined Radio (SDR). When compared with an optimized cycle-accurate VP of MuSIC on a variety of tests, including a multiprocessor JPEG encoder, the accuracy is within 20%, with speedups from 10x to 1000x.

1. INTRODUCTION

1.1 Related Work

Traditional instruction set simulators and system-level virtual prototypes are accurate, but can be slow and difficult to modify. Compiled code simulators such as those from VaST[3] significantly increase performance, but are time consuming to create and modify, and may be still too slow for large multiprocessor applications. The POLIS project [4] "synthesizes" source code from a formal specification, called Codesign Finite State Machines (CFSMs). It features performance estimation [9] based on CFSMs and S-Graphs, an intermediate representation used in the synthesis process. For simple processors this approach had a maximum error magnitude of 25%, but it can only be applied to synthesized code from CFSMs, whereas our technique is based on the application code and achieves better accuracy.

Design Automation and Test Europe (DATE) March 2008, Munich, Germany

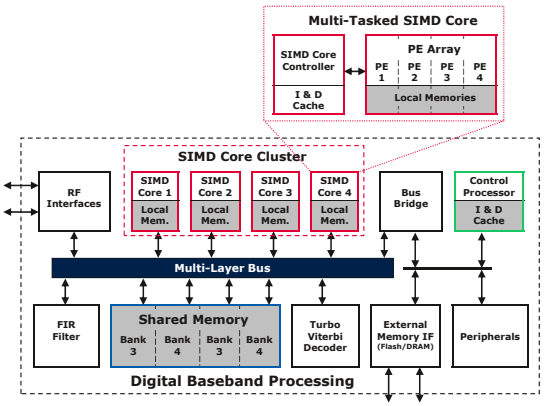


Figure 1: MuSIC System Architecture

In [5] the authors introduce two approaches for performance estimation: object code based and virtual compilation based. Our technique is at a much higher level than both of these, yet it achieves good accuracy and, aside from detecting the synchronization instructions, is independent of the assembly used.

MESH [8] is a high level environment for exploring different multiprocessor system on chip architectures. Performance is based on user specified instruction counts for threads executing on processors. We automatically annotate cycle delays obtained directly from target source code running on the virtual prototype. Further, we could easily extend the tool to output instruction counts.

1.2 Architecture and Models

Figure 1 shows the MuSIC system architecture. It features: four multi-tasked SIMD (Single Instruction Multiple Data) core clusters specialized for signal processing, an ARM processor for control layer processing, and hardware accelerators. All processing blocks can access shared memories via a multi-layer bus. Each task in a SIMD core cluster is controlled by a general purpose RISC processor running a custom multiprocessor RTOS (Real-Time Operating System) called ILTOS, where each RISC processor can run a single thread. Applications can access processing, memory, and communication resources through the ILTOS API.

1.2.1 Architectural Simulation Models

For cycle accurate simulation, the MuSIC team developed an optimized VP in SystemC[1]. The SIMD control processor is simulated by a cycle-accurate model created with Coware's Processor Designer. The VP is parameterizable in

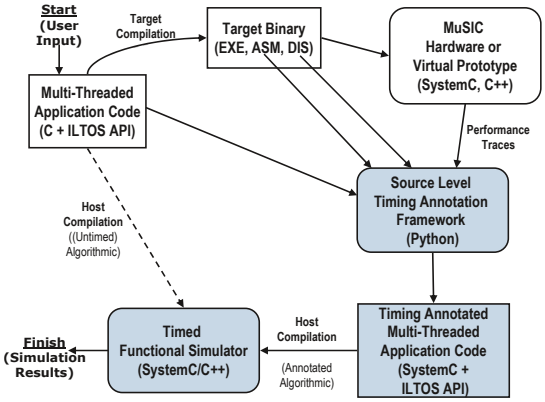


Figure 2: Annotation Tool Flow. Shaded parts indicate the contributions from this work.

a variety of ways including: the number of SIMD processors, enabling/disabling different components, different levels of accuracy and speed for some components, and the generation of statistics and performance traces.

1.2.2 API-Compatible Simulator

For coarse multithreaded algorithm evaluation and early architecture exploration an API-compatible functional simulator was developed. It implements the ILTOS API, and allows applications to be compiled directly to the Windows host and run significantly faster than the VP. However, it has no notion of performance. The simulator was ported to SystemC 2.2 to support timing annotation and tracing. Annotation is done by using a special *DELAY* function, which directly maps to the SystemC timed *wait* function.

1.3 Tool Flow

Figure 2 shows the annotation tool flow. It begins with an application coded for the ILTOS API. This can either be compiled and run in an untimed manner on the functional simulator (as shown by the dotted line), or it can be compiled to the target platform and then run on either the VP or the actual hardware. For annotation purposes the application is run on the VP. Then our framework reads in the the application code, the assembly files, the disassembled object file, and the execution trace produced by the virtual prototype. Based on this information a delay for each line of the original source code is calculated and then added to a timing annotated version of the application. This code can then be compiled and run on the functional simulator to get timing information without simulating the architecture.

1.4 Basic Definitions

An *execution trace* corresponds to a run of a program and consists of the instructions beginning execution on each processor at each cycle. An *instruction* is a single line of assembly at a given program address. The delay of an instruction is calculated by subtracting its start-time from that of its successor. A *block* is a contiguous sequence of instructions in the program that have the same *label*. Labels attach the given instruction or block to a given line number (or null value). The labels are extracted debugging information. A *basic-block* is a block that, once entered[1] is never interrupted with external instructions and always executes its final instruction (note: it can be entered at a midpoint). The delay of a basic-block for a given execution is equal to the sum of the delays of all of its instructions for that execution. A *line* is the set of blocks with the same label. Line delays are explained in Section 2.2.1.

2. SINGLE PROCESSOR ANNOTATION

2.1 Constructing Annotation Structures

First the assembly and disassembly files are unified into a single description based on the locations of line numbers and functions present in the debug information in the assembly language, and the associated instruction addresses and sizes from the disassembled executable. Each source file is split into functions. Each function is split into lines, which are then split into blocks. Then, *jumps* are extracted from the processor's execution trace. A jump occurs whenever an instruction in the execution trace is not directly followed by its successor. The blocks are sliced at these boundaries to form more blocks. This separates the blocks from the functions that they call, which can improve accuracy.

2.2 Calculating Block-Level Annotations

Next, the execution trace is stepped through to calculate the delays for each execution of the basic-blocks. When a block is entered its execution count is increased by 1. When it is exited the block's delay is annotated to its line.

2.2.1 Calculating Line-Level Annotations

When a block's execution is added to its line, their execution counts are compared and if the block's is higher then the line's execution count is incremented. This approximation works for most cases. There are two types of annotations for lines. *Internal annotations* are the cycles and instructions of the blocks associated with that line of code. *External annotations* are the cycles and instructions associated between source blocks, with annotations between lines going with the later line. The internal and external annotations are summed together for each *execution* of the line.

2.3 Generating Annotated Code Source Code

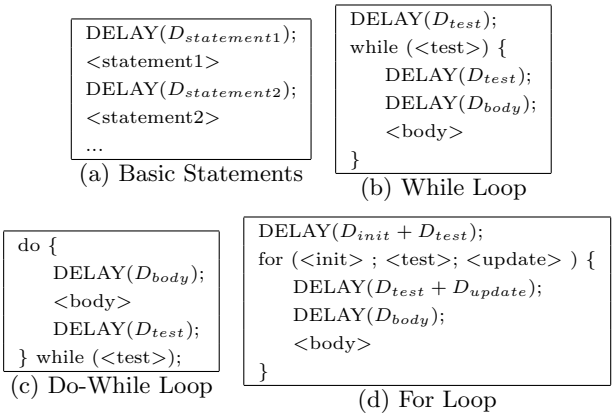
The delay annotations for each line are calculated based on the average cycle count[2] over its executions. Figure 3 shows examples of the different annotation cases. In this figure, D_N represents the delay for statement <N>, and the DELAY function is the annotation. For the general case a line of code has its delay written directly before it in the annotated source file. Figure 3(a) shows the basic annotation of the statements <*statement1*> and <*statement2*>.

Figure 3(b) shows an annotated *while-loop*. In it <*test*> represents the test condition and <*body*> is the loop body. While-loops are detected by examining each line of the source code for lines that start with whitespace immediately followed by the *while* keyword. In this case, the test delay is annotated before and after the *while* loop statement. Interestingly, *do-while* loops do not need special handling, because in the test condition and body execute the same number of times. Figure 3(c) shows an annotated do-while loop.

Figure 3(d) shows an annotated *for-loop*. Like the while-loop, the for-loop has a test condition called <*test*>, but it adds initialization and update statements respectively named <*init*> and <*update*>. For-loops are detected in the same

[1]A basic-block can be entered at a midpoint.

[2]Maximum and minimum cycle counts can also be used.



(a) Basic Statements (b) While Loop (c) Do-While Loop (d) For Loop

Figure 3: Annotation Examples

manner as while-loops, but with the *for* keyword. The initialization statement will always be the first block executed in a for-loop. Therefore, this block's delays are subtracted from the delays of the for-loop's line delay. The first block's delay is annotated above the line, and the remaining delay is annotated directly below it.

3. MULTIPROCESSOR ANNOTATION

Multiprocessor timing annotation is very similar to the uniprocessor annotation. It applies the same techniques, but it combines the line-level annotations from every processor to calculate the overall annotations. Also, two special cases and their handling are detailed below.

3.1 Startup Delays

Without special handling the startup delay before any user code is executed is assigned as a regular delay to the first executed line of user code. If this line executes on more than one processor, then the delay will be added each time instead of just the first time. To deal with this a static variable called "started_up" is included in the functional simulator. For all other threads the startup delays are ignored. The startup delay is guarded for on the first source line executed in the below manner:

```
if (started_up == 0) {
  started_up = 1; Delay_Thread(<D_startup>);
}
```

3.2 Inter-Processor Communication

Since the line-level annotations are calculated independently for each processor, inter-processor communication is counted for both processors, leading to double counting. One way to deal with this is to analyze all of the processor traces concurrently, but this approach would require excessive storage, and that the annotation framework understand the multiprocessor functions and their interactions. Instead, communication is handled by ignoring ILTOS API function delays and then replacing them with pre-characterized delays during simulation. For the results, *direct-measurement* refers to directly measuring all delays, and *characterization-based* uses the above described technique.

4. RESULTS AND ANALYSIS

The annotation algorithms were implemented in Python 2.5 for the SIMD control processors in MuSIC. The code for reading in the different files is specialized, while the core algorithms are implemented generically. All of the experiments were run on a 2.0 GHz Core Duo laptop running Windows XP with 1 GB of memory. The applications targeting the control processors were compiled without optimization. The annotated source code was compiled with Microsoft Visual C++ 2005 and linked to the timed functional simulator. The annotated source code running on the timed functional simulator was compared to the the virtual prototype in terms of speed and accuracy for the same data and for different data. For non-disclosure purposes the numbers are given in a relative manner.

Table 1: MiBench Tests

Benchmark	Small Results Error (%)	Small Results Speedup	Large Results Error (%)	Large Results Speedup
adpcm.encode	-1.67%	16.3	-0.70%	16.3
adpcm.decode	-2.25%	15.6	-1.31%	40.1
dijkstra	-12.63%	25.7	-17.46%	27.5
patricia	-0.47%	81.6	-1.18%	65.5
rijndael.encode	-1.26%	129.9	-2.96%	229.6
rijndael.decode	-6.52%	159.1	-1.69%	234.5
sha	0.00%	1,030.8	0.00%	984.4
stringsearch	3.85%	14.6	-13.95%	28.7
avg. magnitude	2.80%	184.2	4.91%	203.3
max. magnitude	12.63%	1,030.8	17.46%	984.4

Table 2: Multiprocessor Tests

Benchmark	Thread Count	Direct Measurement Error %	Characterization Based Error %	Characterization Based Speedup
message test	3	0.00%	0.21%	80.1
streaming_test	4	48.77%	0.18%	207.5
thread_test	19	767.24%	-2.24%	526.9
JPEG_multi	5	93.45%	-7.39%	17.0
avg. magnitude		227.37%	2.50%	207.9
max. magnitude		767.24%	7.39%	526.9

4.1 Results with Identical Data

4.1.1 Single Processor with Identical Data

The annotator was evaluated on ten single-processor tests from the ILTOS library, using both the direct-measurement and the characterization-based annotation techniques. The maximum error magnitude of the two were 0.5% and 1.6% respectively. The speedup for these tests ranged from 11x to 139x, and averaged 47x.

The annotation was also tested on eight benchmarks from the MiBench benchmark suite [7] that were easily ported to the control processors. Table 1 shows the results using characterization-based annotation. The first set of results are for the benchmarks running on the small data set, and the second set of results are for the large data set. The accuracy is within 18%, and for most cases is within 3%. There is a wide range of speedup between 14x and 1030x.

4.1.2 Multiprocessor with Identical Data

The annotator was run on four multiprocessor tests. *Thread_test*'s main thread creates a new thread running the same code and then waits for it to finish. This creation and waiting happens until each of the 19 control processors in the system is allocated one thread. Then the threads terminate one by one. *Message_test* and *streaming_test* feature inter-thread communication. *JPEG_multi* is a five-threaded JPEG encoder ported from PThreads.

Table 2 shows the results for the multiprocessor examples,

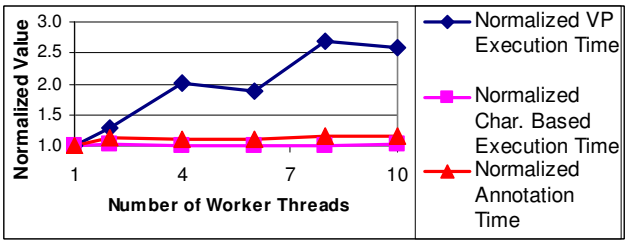


Figure 4: Normalized execution and annotation times for streaming test with varying numbers of worker threads

with the third and fourth columns indicating the accuracy for direct-measurement and characterization-based annotation respectively, and the last column shows the speedups. For these tests, the direct measured approach has average and maximum error magnitudes approximately 100x times larger than those from the characterization-based approach.

4.2 Results with Different Data

The results so far mentioned are so accurate because they are annotated with performance measured from the application running on the same data. To evaluate the performance better, it is necessary to compare the accuracy of the annotated code running on different data and different control flows than those used for training.

4.2.1 Single Processor with Different Data

We ran the Dhrystone benchmark with loop counts of 1, 10, 100, 500, and 1000 to obtain annotated source files. Each annotated result was run on all of the loop counts to evaluate the data-dependence accuracy of the annotation. The single iteration results have an average error magnitude of 25% and a maximum error magnitude of 47%. These errors are so large because they are based on the initial execution where the code is loaded into the instruction cache. The last three loop counts are much better, with average error magnitudes under 2.1%, and maximum error magnitudes under 3.5%.

For all of the mentioned MiBench tests except for stringsearch (which initializes data in the source code), the large trained annotated code was tested on the small data sets, and vice versa. For the large trained code the average and maximum magnitudes were 4.6% and 17.5%, and the magnitudes were 2.9% and 12.6% for the small trained code.

4.2.2 Multiprocessor with Different Data

Then streaming_test was run on files with 15, 20, 40, 100, and 500 elements. The generated annotated source files for each number of elements were then run on all of the numbers of elements. The full results had 0.7% maximum error magnitude. JPEG_multi was evaluated on images of different sizes and its error magnitude stayed below 8.2%.

The number of worker threads in streaming_test were then varied from 1 to 10 for a file with 500 elements. These had a maximum error magnitude of 0.26% when running on the same data. Figure 4 shows the normalized run times for it and shows that the annotated code scales significantly better than the VP. All of the generated source files were then tested on all of the worker thread configurations, and had a maximum error magnitude 3.03%. The speedup for the one worker was 46x.

4.3 Annotation Runtime

The annotation framework's runtime is linear in the sum of the sizes of the input files, with the execution trace's length dominating. Due to space constraints the execution trace is preprocessed and compressed with gzip. Then, each processor's execution trace file is read in and the annotations are calculated for it. Based on measurements the runtime ranged from 0.5x to 5x that of the VP's runtime, with over half of this is taken by the preprocessing. The use of gzip and Python significantly slow things down. The annotation framework's runtime could be greatly reduced if it were integrated to run concurrently with the VP.

4.4 Limitations

Currently the framework does not fully parse the original C application code. It makes some assumptions on the syntax, which are currently resolved by using a source-code beautifier [2], and by changing offending statements. Furthermore, its accuracy can be impacted by putting multiple commands on the same line. Also, annotations illegally placed are presently fixed manually.

All of the experiments are run without compiler optimizations. Using compiler optimizations makes obtaining accurate annotations more difficult. This also impacts debuggers. Approaches such as [10] address it.

5. FINAL WORDS

Automated timing backwards-annotation at the source code level was presented for a heterogeneous multiprocessor. While the implementation was for a specific architecture, the framework is highly generic and quite portable. This technique achieved accuracy within 20% for single and multi-processor applications running on the same data. As expected, the annotations were not as accurate for different data. The annotated code was 10x to 1000x faster, with multiprocessor programs and programs calling complicated external libraries exhibiting the greatest benefit.

While the timing annotation works well for certain classes of programs, such as the ones that fit into cache and have very regular communication patterns (e.g. streaming applications), it does not take into account resource contention or communication overhead. A promising path to solve this problem is extending annotation to handle memory and communication traffic.

6. REFERENCES

[1] Open SystemC Initiative Web Site: http://www.systemc.org.
[2] Uncrustify web site: http://uncrustify.sourceforge.net/.
[3] VaST Website: http://www.vastsystems.com.
[4] F. Balarin, et. al. *Hardware-software co-design of embedded systems: the Polis approach.* Kluwer Academic Publishers, Boston; Dordrecht, 1997.
[5] J. Bammi, et. al. Software performance estimation strategies in a system-level design tool. *Proc. of CODES*, pages 82–6, 2000.
[6] H.-M. Bluethgen, et. al. A programmable baseband platform for software-defined radio. In *Proceedings of SDR FORUM*, 2004.
[7] M. Guthaus, et. al. Mibench: A free, commercially representative embedded benchmark suite. *Proc. of the 4th Workshop on Workload Characterization*, pages 3–14, 2001.
[8] J. M. Paul, et. al. High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors. *ACM Trans. Des. Autom. Electron. Syst.*, 10(3):431–461, 2005.
[9] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. *Proceedings of the DAC*, pages 605–610, 1996.
[10] L.-C. Wu. *Interactive Source-Level Debugging of Optimized Code.* PhD thesis, University of Illinois at Urbana-Champaign, August 1999.