An Independent Evaluation of the

picoChip PC102
Software Development Tools and Programming Model

By the staff of

BDTi INSIGHT • ANALYSIS • ADVICE ON SIGNAL PROCESSING TECHNOLOGY

OVERVIEW

picoChip is a fabless semiconductor company that offers high-performance chips for computationally demanding wireless infrastructure applications. picoChip's chips contain hundreds of processors connected in a MIMD (multiple instruction, multiple data) configuration.

One of the key challenges for massively parallel chip vendors like picoChip is that the software development process for these chips tends to be significantly more complex than that used for single-processor chips. To address this challenge, picoChip has developed a programming model and associated tool suite that are intended to help reduce application implementation effort and complexity.

In this white paper, BDTI, an independent analysis company focused on digital signal processing (DSP) technologies, evaluates the tools and design methodologies used to implement applications on a picoChip chip, the PC102. We explore how well these tools and methodologies help mitigate the complexity of the chip and allow the programmer to create robust, efficient application implementations.

Contents

Introduction

picoChip sells high-performance multi-core chips for wireless infrastructure applications. The company was founded in 2000, and has been shipping chips since 2002. In 2004 picoChip introduced the PC102, which contains 308 programmable processors and 14 co-processors. The PC102 and other picoChip products are programmed using picoChip's software development tool suite, called the "picoTools," which includes some tools that are similar to those used in programming traditional DSPs and other types of processors (such as a compiler, assembler, and simulator) and some that are different—such as a specialized place-and-route tool and a graphical design viewer.

BDTI recently completed an evaluation of the picoChip tool chain using the PC102 chip, and this paper presents the results of our evaluation.

About BDTI

Berkeley Design Technology, Inc. (BDTI) is widely recognized as a trusted source of independent analysis of processing engines and tools targeting embedded applications. BDTI uses its signal processing benchmark suites and in-house expertise to evaluate the signal processing capabilities of various processing engines, including DSP processors, general-purpose processors, FPGAs, and multi-core devices. BDTI also provides contract software development services, and has completed numerous embedded

software projects on a range of processing engines. For further information see www.BDTI.com.

Evaluation Methodology

The most effective way to evaluate software development tools is to use them. The key is to choose a test project that is complex enough to exercise the tools in a realistic manner, but not so complicated that it is prohibitively time-consuming to implement. For this project, we chose to implement an FFT (fast Fourier transform), because it is computationally demanding and widely used in signal processing applications, including those targeted by picoChip's PC102. BDTI has implemented variants of this algorithm on dozens of different processing engines, and we've found that although it is more complex than, for example, an FIR filter, it is not overly time-consuming to implement, even on a complex processor architecture.

BDTI began the picoChip tools evaluation project with a strong background in developing signal processing applications for a wide variety of processing engines, but with no direct experience using picoChip's products or tools. Thus, we were knowledgeable novices, as is likely to be true of many of picoChip's customers.

Our goal was to use the FFT as a test project for assessing the picoChip tools and software development paradigm. We were specifically interested in evaluating the following characteristics:

- **Conceptual complexity.** How hard is it to understand the picoChip multi-core architecture and associated development methodology?
- **Ease-of-use.** Are the tools fairly straightforward and robust?
- **Quality/efficiency of implementation.** Do the tools enable programmers to create implementations that are efficient (in terms of speed, power consumption, and resource usage)? Are the debugging capabilities effective?
- **Documentation.** Is the documentation well organized, easy to read, and clear?

The process of implementing and testing the FFT on the picoChip PC102 provided us with hands-on experience with the chip's tools and programming model, and allowed us to assess most of the characteristics listed above.

There was, however, a drawback to our choice of a test project. The FFT is relatively small and reasonably well structured and, as such, does not allow us to explore how difficult it is to partition a complex application across multiple processors in a picoChip device. For a massively parallel chip like the PC102, this step will often constitute a key part of the overall application development process. To help address this challenge, picoChip provides implementations of several communications applications. Some picoChip customers will use these reference designs as-is; others will use them with modifications; and others won't use them at all. The needs and experiences of these three groups are going to be quite different.

To augment our analysis in the area of application partitioning, we interviewed three system developers who have developed complex applications (e.g., WiMax and wideband CDMA systems) on the PC102. We asked about their experiences in using the PC102, and have included highlights from their responses in our evaluation.

The picoChip PC102

The PC102 is based on picoChip's "picoArray" architecture, which consists of many independent processors that operate in a MIMD fashion. Each processor executes its own instruction stream and processes its own data. The number of processors included in the picoArray depends on the specific chip. The PC102 contains a total of 308 processors, which are called "AEs" (for "array elements").

All of the processors in the PC102 are 16-bit, three-way long instruction word (LIW) RISC processors with Harvard memory architectures, and all have their own local memory. On the PC102, there are three distinct processor types: standard, memory, and control. The processor types vary in their amount of local memory and computational resources. Standard processors have a MAC unit and 768 bytes of internal memory, and are intended for computationally demanding signal processing. Memory processors exclude the MAC unit but bump up the local memory to 8,704 bytes. These processors are intended for memory-intensive portions of the application, such as buffering. The control processors also exclude the MAC unit but have 65,536 bytes of local memory. These processors are intended for control-intensive portions of the application. A PC102 chip contains 240 standard processors, 64 memory processors, and 4 control processors, reflecting the mixture of processing that picoChip believes will be typical in its target applications. All three processor types use the same RISC instruction set, except that MAC instructions can only be executed on standard processors. With the exception of loads and branches, all instructions execute in a single cycle.

The processors' 64-bit LIW contains three execution slots: one for ALU operations, one for load/store operations or a second ALU operation using a second ALU, and one for branch or MAC operations. Using all three slots, up to three operations can be executed in parallel in each cycle, though there are limitations on the combinations that are allowed. Each processor can only access its own internal memory, and communicates with other processors using input/output data ports. Processors are connected via 32-bit "picoBuses" and programmable bus switches. The picoBus connections between processors are defined (using a subset of VHDL, called "picoVHDL") at compile time, so there is no real-time bus arbitration. Once a program is

mapped to the processor array, program execution timing is deterministic.

In addition to the processors described earlier, the PC102 contains 14 application-specific co-processors that are designed to accelerate common DSP/communications tasks, such as Viterbi decoding. These co-processors were not used during our tools evaluation.

Implementing Applications on the PC102

In general, implementing an application on the PC102 requires the following steps:

1. Partition the application into "sub-blocks," each of which will run on one (or sometimes more than one) processor on the PC102. This is a manual process, and requires the designer to functionally decompose the application and map it to the picoChip processor array. The functional decomposition will attempt to balance the processing load across the array for optimal performance, which is typically an iterative process as more accurate estimates of individual processor performance are obtained through the software development process.
2. Define the input/output bandwidth, data types, and bus connections for the sub-blocks. This is accomplished using a VHDL-like language called picoVHDL; the software engineer creates a text file with lines of VHDL code describing the interaction of sub-blocks.
3. Implement the software for each processor. Software can be implemented in C, assembly code, or a combination of the two.
4. Simulate and debug the software on each processor.
5. Simulate the complete application to verify inter-processor communication.
6. Map software to specific processors on the chip. Up until this point, the processors on the chip have been treated as essentially interchangeable, but at this point their relative location is taken into account in an attempt to optimize inter-processor communication and resource allocation. (picoChip refers to this step as "place and switch," and it is conceptually similar to the place-and-route step that is used with FPGAs). As we discuss below, this step is done automatically by one of the tools in the picoChip tool suite.

The implementation process is accomplished using picoChip's tool suite, called the "picoTools." Tools in the suite include:

- picoAnalyze: After the software engineer has defined the bandwidth and data types for each of the sub-blocks and has written the associated software (steps 3-4 above) this tool checks the picoVHDL/ASM source files for syntax errors. Files without errors are placed into a library for further processing.

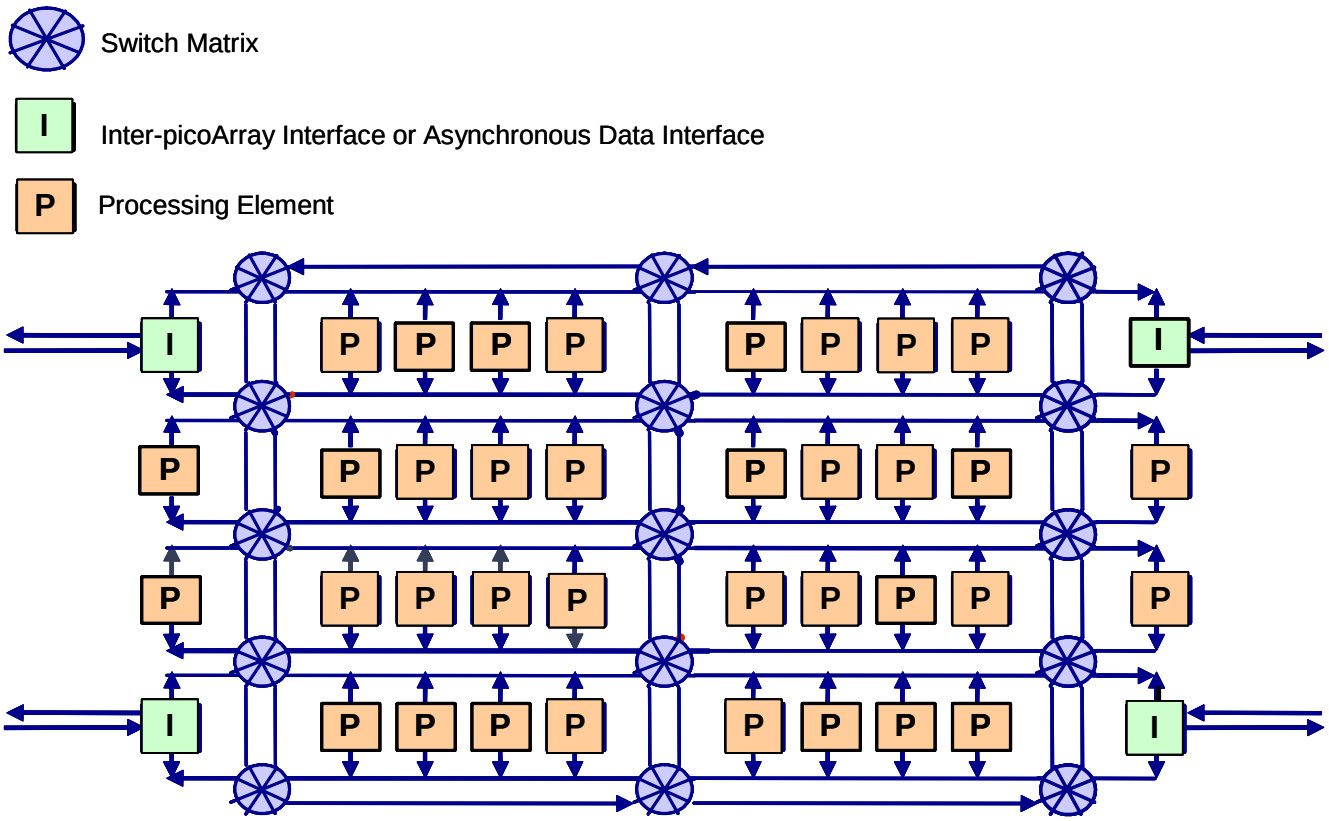

FIGURE 1. picoChip PC102 Key Architectural Features

- picoElaborate: This tool takes the library generated by picoAnalyze and produces a text file with configuration data for each processing element, including compiled code memory images for instruction and data memory, and configuration data for setting up interconnects. It checks for syntax errors, and if none are found, it compiles and/or assembles the code.
- picoGcc: This is the picoChip C compiler. If a processor is programmed in C, picoElaborate will invoke picoGcc to compile the C code.
- picoPartition: This tool maps the sub-blocks and their connections to a specific device (e.g., the PC102).
- picoPlastic: The tool maps sub-blocks to specific processors on the chip, and tries to optimize inter-processor communications and processor location (i.e., placing blocks in processors that are near needed resources, such as I/O or peripherals). It is analogous to the place-and-route tools used for FPGAs.
- picoDebugger: picoChip's debugger includes a cycle-accurate simulator, and can also be used with the hardware. The design can be viewed using the "Design Browser," a separate graphical tool that shows a block diagram representation of the full implementation. The user can click on a block to drill down and view the associated software.

A detailed view of the design process and point tools used in the design of a picoChip PC102 application is given in Figure 2.

The picoTools run on Linux and are not available for Microsoft Windows. They can be invoked from the command line, or from a graphical front-end, called "picoDeveloper."

Implementing the FFT

Now that we've introduced the basic software development process and tools, we'll describe our experiences implementing the FFT on the PC102 following the process outlined above.

Partitioning the Application

To run on the PC102, an application must be partitioned into sub-blocks, each of which will run on a separate pro-

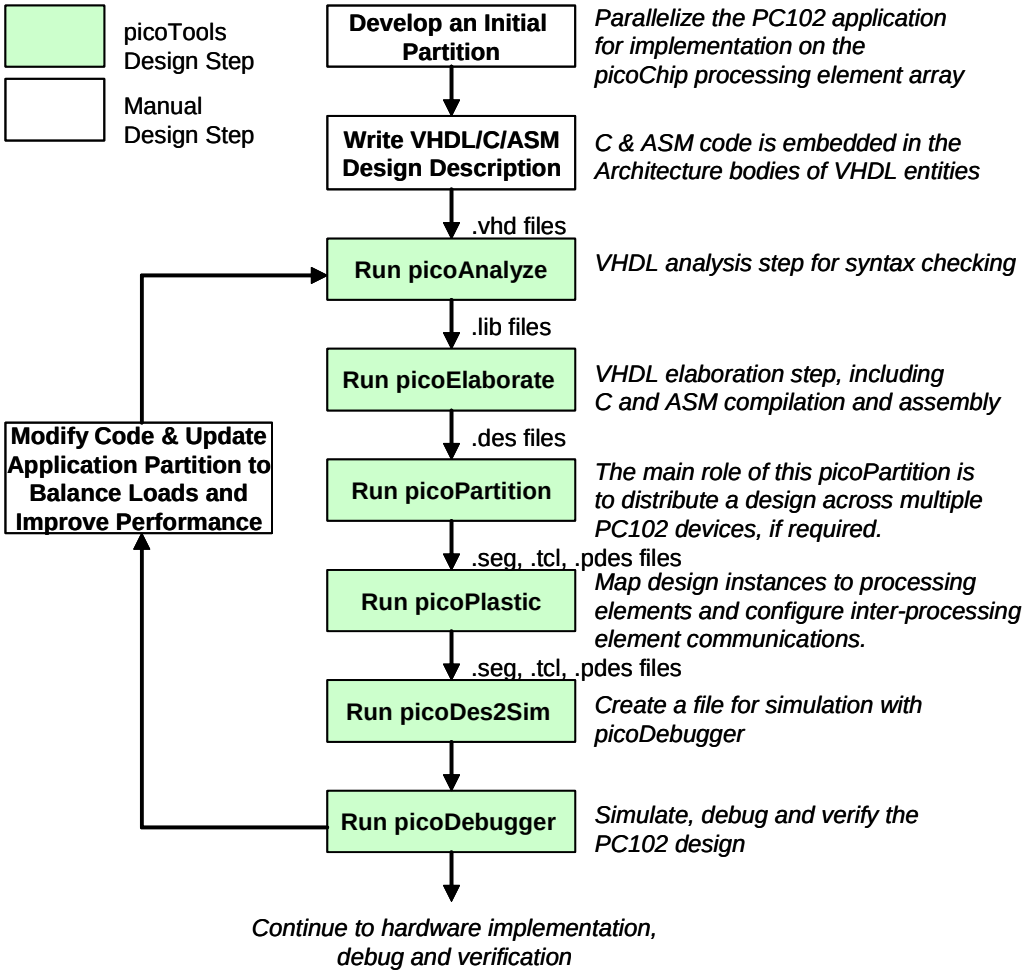

FIGURE 2. The Design Process for a picoChip PC102 Application

cessor. Partitioning the application in this way is done manually by breaking up the design into roughly balanced computational loads based on estimates of the number of cycles each sub-block will take to execute. In our example, we developed a partition that separated the eight "passes" of the FFT into eight sub-blocks; each pass being largely independent of the others and suitable for in parallelization.

As part of the partitioning process, we needed to evaluate which type of processor to use (i.e., the standard, memory, or control processor). We concluded that each pass required a standard processor, because this is the only processor that includes a MAC unit. We needed the MAC unit for two reasons:

1. The FFT twiddle factors are multiplied by input data.
2. For the add/subtract operations used in a radix-2 butterfly, the accumulator in the MAC unit was used to provide greater dynamic range than the 16 bits provided by general purpose registers (A register pair could also have been used to achieve the same dynamic range, but would have required more cycles.).

Once we decided to use a standard processor for each pass, we needed to design the data flow into and out of each pass. With single-core DSPs, a frame of data stays in one location in memory while being processed. For the PC102 implementation, however, the data needs to flow from one processor to the next, because memory is not shared between processors.

In order to keep the data flowing between processors, we needed a "ping-pong" buffer to hold the frames. (For a 256-point FFT, the frame size is 256 complex data samples, where each complex sample requires 4 bytes.) While one frame was being processed, the other was being buffered. When one frame finished, the two buffers would be switched. We also needed memory to store the twiddle table. As a result, the standard processor's 256 bytes of internal data memory were insufficient to meet the memory requirements of each pass; we had to pair each standard processor with a memory processor. Furthermore, we needed an additional memory processor at the end of the passes, to be used as output buffer.

Thus, the total number of processors used to implement our FFT was 17 (out of 308)—two for each of the eight passes, and one for the final buffering step. However, after we implemented this version we determined that it would have been possible to further optimize the implementation to use 13 processors (six memory processors and seven standard processors). Pass eight, the last pass, has no twiddle multiplication and therefore doesn't need a standard processor, so it could be implemented on a memory processor and merged with the output buffer. Furthermore, because the memory requirements decrease by 50% for each pass, passes six and seven could each be implemented with a standard processor alone. Because we were primarily interested in evaluating the tools rather than achieving maximum throughput, we did not implement this optimization. It should also be noted that in a recent implementation of the BDTI Communications Benchmark (OFDM)™ the same FFT implemented using radix-4 butterflies was implemented in 5 processors. The radix-4 implementation represents a more efficient solution and a better design choice for this FFT on the picoChip PC102 than the radix-2 solution used as the design example in this paper.

Overall, we found that partitioning our simple FFT application required some thought but was fairly straightforward, though as noted above, we did discover later in the process that we could have re-partitioned the application more efficiently. We expect that, for applications that are larger, more complex, and less regularly structured, the partitioning step is likely to be quite challenging, and will probably require multiple iterations to create efficient implementations. Furthermore, for applications where key parameters change over time (e.g., sampling rates), it will be difficult to define a suitable partitioning scheme that doesn't leave much of the chip's performance on the table.

As described earlier, we interviewed three developers from different companies who have implemented complex applications on the PC102, and asked them how difficult it was to partition their application. Two of them started with WiMax reference designs from picoChip and made modifications. For those two vendors, the partitioning scheme had already been defined by the reference code, and they did not have to perform this step themselves. The third developer was developing a different kind of application, a multi-antenna signal processing algorithm. This developer stated that partitioning the application was a challenging part of the application development process and consumed a significant portion of his application development time.

Writing the Software

Having partitioned the FFT into sub-blocks and mapped these sub-blocks to specific types of processors, we began writing the software. The first step in this process is to create a "structural instance" of each sub-block using picoVHDL. PicoVHDL is a subset of the standard VHDL language, augmented with several extensions by picoChip. The structural instance defines the inputs and outputs (in terms of bandwidth and data type) for each sub-block.

The processors on the PC102 can be programmed in C or assembly language, or a combination of the two. As described earlier, they are three-issue, long instruction word (LIW) processors that use a RISC instruction set plus a number of specialized instructions for DSP and communications applications, such as add or subtract with saturation, arithmetic left or right shift with saturation, and bit reverse. With the exception of memory accesses and branches, all instructions have single-cycle latencies. This simplifies assembly-level coding, debugging, and optimization, and makes it relatively simple to ensure real-time behavior. The

MAC unit on the standard processors has two 40-bit accumulators, but does not provide hardware support for fractional multiplication, saturation, or rounding. These capabilities are commonly required in signal processing algorithms, and must be implemented in software on the PC102.

To implement our FFT, we used primarily assembly code. We found that, although picoChip's C compiler (picoGcc) generated reasonably well-optimized code, assembly code was needed to pack the real and imaginary components of the complex data into a single register (to improve performance), and make use of specialized instructions such as bit-reversal. Using assembly code also made it easier to predict cycle counts, which is useful for evaluating performance and optimizing the code. Overall, we found the individual processors easy to program, and our experience was echoed by the three vendors we interviewed.

In preparation for verifying the functionality of the code, we generated intermediate fixed-point test vectors for each of the eight passes using FFT reference C code executing on a PC. We then used picoVHDL to build a test bench with file I/O. The input to the test bench was the input test vector for the current pass; the output from the test bench was verified against the corresponding output test vector.

As described earlier, we implemented each FFT pass (which were similar, but not identical) with one standard processor and one memory processor. The memory processor buffered the input data from the test bench and then sent the data pairs and twiddle factors to the standard processor. The standard processor computed the radix-2 butterfly and sent its output to the test bench, which wrote the output to a file for verification against the test vectors.

Our performance target for the FFT was taken from the definition of the BDTI Communications Benchmark (OFDM)™, which requires a 256-point radix-2 butterfly to finish in 32 cycles. Theoretically, this target could be achieved fairly easily by using two or three LIW slots for portions of the code. However, an added complication of programming the PC102 when compared to programming single-core processors is that provision must be included in the code for data transfer between processing elements. For the FFT, this meant that additional development effort was required to write the code that managed and synchronized data transfer between FFT stages. By careful scheduling of operations in LIW instructions we were able to minimize any performance overhead associated with data transfer and synchronization. Once we had written the picoVHDL/assembly software, we used the picoAnalyze tool to generate a library.

Mapping Software to Processors, Debugging.

After we ran picoAnalyze, we ran picoElaborate to generate the machine code for each instance. In addition to compiling the code for each processing element, by using the -write_statistics command line option, picoElaborate generates a file that provides information on the configuration of picoChip device resources used in the design. For each resource, this information includes the type of resource, memory usage, LIW and single instruction counts, code size and density, and LIW instruction efficiency. An

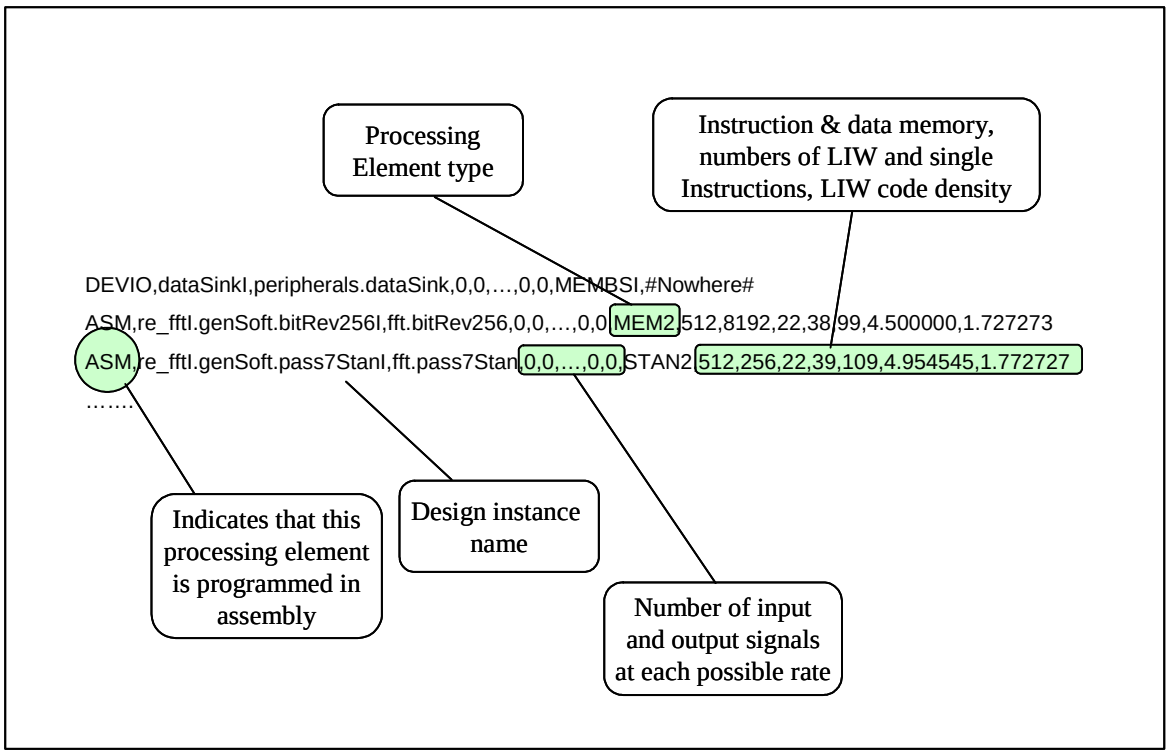

FIGURE 3. Statistics File Format Generated by PicoElaborate

example of the statistics file for our design example is shown in Figure 3. The next step in our design flow was to run picoPartition. The real role of picoPartition is to partition a design across multiple picoChip devices. Since our example uses only a single device, picoPartition simple allocates instances to that device.

Finally, we used picoPlastic to place all specified processors onto the PC102 picoArray and to build the bus connections between them. Once this step is completed, the design can be run and debugged in hardware. Unlike multi-threaded programming models, program flow on the PC102 is deterministic which simplifies the debugging process. PicoPlastic generates an array display diagram that provides a graphical display of how the design is implemented on the device (Figure 4).

For our example, we ran picoAnalyze, picoElaborate, picoPartition, and picoPlastic as command-line tools included in a linux BASH script.

Tool Strengths and Weaknesses

We kept careful notes as we implemented the FFT, and here we share our assessment of the strengths and weaknesses of the tools.

STRENGTHS:

- Installing the picoTools was easy. The toolset is downloaded as an RPM ("Red Hat Package Manager") file, and can be installed like any other RPM file. License installation is also easy—at least for the evaluation license we used. After the installation is finished, some environmental variables need to be set up. Everything is described in detail in the document, "System Requirements and Installation Guide."
- The toolset is robust. It crashed only once during our two-month evaluation period. (We didn't test it using PC102 hardware, however.)
- The tools are quite fast. Building requires a couple of minutes, and simulation was also very quick. We should note, however, that because our test program was small, the processing demands made on the tools were relatively modest compared to what they would be for complex applications that used more of the chip's processing array. According to picoChip, a full WCDMA design (on a single PC102 device) that uses 233 programmable processors and 561 signals takes 205 seconds for the full build process, including Place and Switch. BDTI has not verified this data.
- The tools provide pre-defined "probes" that are intended to be used for hardware debugging. A probe is a processor with the sole purpose of capturing data for debugging. The probes are non-invasive and do not affect the timing or operation of program execution. Data can be captured and sent to a host processor or saved (with or without timestamps) for further investigation. The pre-defined probes have 8-cycle resolution, which should be sufficient for most debugging pur-

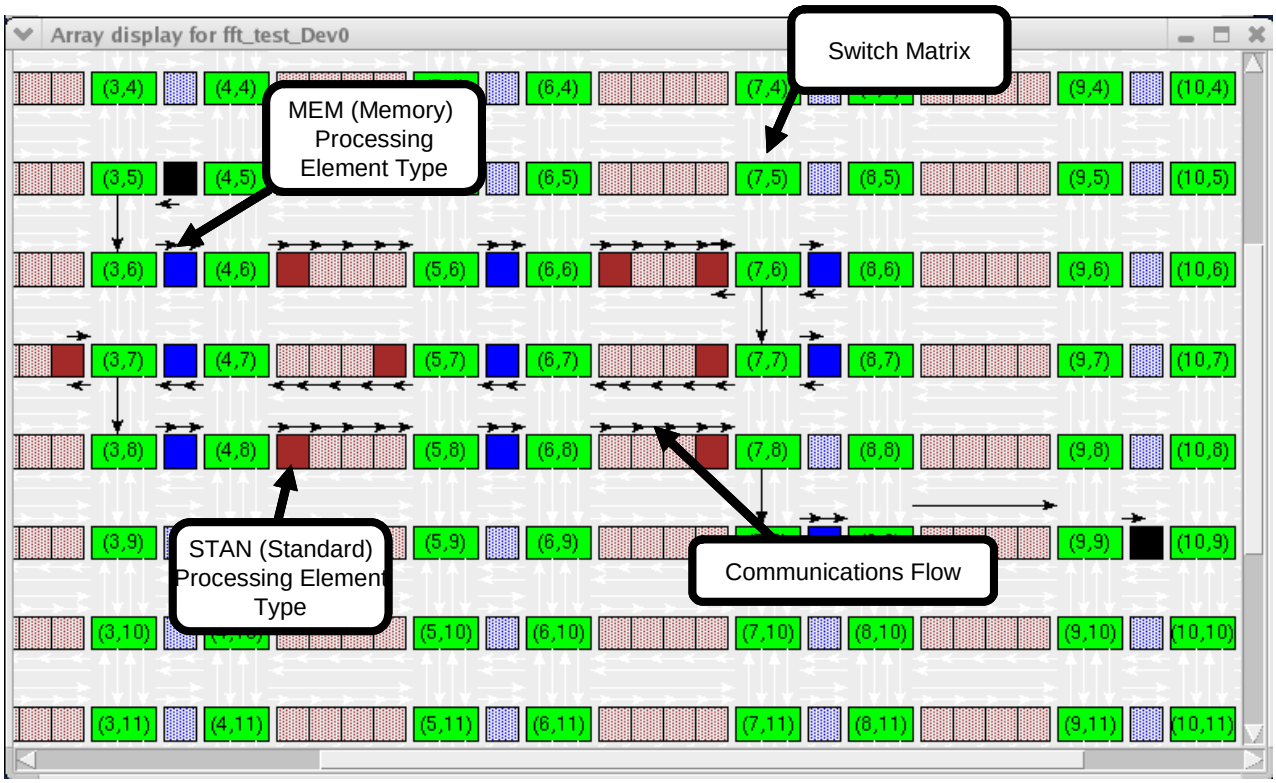

FIGURE 4. Array Display Diagram

poses. Adding probes into a design is simple; probes can be added without changing the original design's source code.

- The automatic placing and routing of processors on a device works well and generally doesn't require the developer to do any additional analysis or optimization.
- Support for Tcl scripting in each tool in the picoTools toolset provides a powerful degree of flexibility to the user. Tcl scripting can be used either in GUI-based or command-line execution of the tools. In addition to support for standard Tcl commands, picoChip provides custom commands for each tool. For example, the debugger has a full set of Tcl commands for debugging, such as single-stepping, running, breakpoint enabling/disabling, getting/setting instance registers or memories, profiling, reading cycles, opening a log file, etc. Because each tool interacts with Tcl commands, it's easy to write scripts to perform repetitive tasks.
- The Design Browser is useful for visualizing the design. It provides a graphical and text-based hierarchical view of the structure of the design implementation.
- PicoPlastic generates a diagram with detailed information about how the design is mapped onto the chip. By clicking any item, a description about its property and instanciation will be displayed.
- In general, the picoTools documentation is well-organized, well-written, and provides thorough explanations.

WEAKNESSES:

- The Tcl shell lacks support for command-line editing. Typing commands within the shell is tedious and error-prone.
- While the debugger is very powerful when used with the Tcl shell, its GUI provides only limited functionality. For example, the GUI debugger should be able to display data in a processor's internal memory as a waveform. But even though the debugger can dump data into a file, the waveform viewer provided with the toolset can't display the waveform properly. It would be helpful if it were possible to profile an algorithm before partitioning it, but since the profiler works at the processor level, it can only be used after partitioning. In general we found that the traditional command line, makefiles, and scripts are much more efficient and powerful for real application development than using the GUI-based versions of the tools.
- It would be helpful if the documentation listed all of the assembly instructions and the Tcl commands in the table of contents. It doesn't, however, so the developer has to read through many pages to find a specific instruction or command.

Observations and Conclusions

Like the tools for other massively parallel chips, the picoTools don't support automatic algorithm partitioning—which, as we discussed earlier, is probably the hardest part of developing an application for a massively parallel device. To be most effective, users should manually parallelize their algorithm before implementing on the picoChip device in VHDL and C. Many users, however, will be starting from a C implementation of their application. Since C is inherently a sequential language, functionally decomposing an algorithm implemented in C into parallel sub-blocks can be challenging, especially for complicated algorithms with data inter-dependencies. It can also be challenging to find a partitioning scheme that balances the workload across processors. We didn't evaluate this process ourselves in any detail since our test application was simple enough to enable fairly easy partitioning. Of the developers we interviewed, only one had developed an application from scratch and performed the partioning step (the other two modified reference designs from picoChip and did not need to do the partioning themselves.) This developer characterized the process as difficult and said that he spent a significant amount of time on this step. In general, we expect that the level of difficulty associated with partitioning will depend on the complexity of the application. And customers who use picoChip's reference designs may not need to delve into partitioning at all.

On the PC102, the total execution time for the application is the time taken by the slowest sub-block—a single processor can stall the whole chip. If this happens, either that sub-block or the whole algorithm will probably need to be re-partitioned. Often, more processors will need to be recruited to achieve the desired speed, resulting in fewer channels per chip. For complicated algorithms that are not well structured, one might need to re-partition several times to find the best balance. On the PC102, algorithm-level partitioning is likely to have a much greater impact on performance than optimizing at the processor level. This means that to some extent the focus of the optimization process is shifted upwards, away from the traditional hand-optimization of C or assembly code, though C/assembly optimization will still often be required.

At the C/assembly level, we've found that the processors on the PC102 are relatively straightforward to program. More generally, once the application has been designed and partitioned, the tools provide good support for implementation and debugging, though we would prefer to see a more sophisticated GUI. For our FFT test case, the picoTools were fast, robust, and reasonably intuitive. Our overall experience with the picoChip tools and programming model was positive—they appear to be well suited for the applications they target.