tensilica

WHITE PAPER

Boost ASIC and SOC Performance by Matching Processor to Task through Automated Processor Generation

System architects face a number of important design decisions when creating the best ASIC or SOC structure for a target application. Good design choices early in the design process reduce silicon cost and power, increase system performance, shrink development time, and improve verification efficiency. This white paper examines the major decisions that must be made when architecting ASICs and SOCs and guides the designer to a systematic approach for developing design structure using processors as the fundamental building blocks of the SOC. This design flow encourages wide use of processors as the default for implementing on-chip tasks and focuses on how to balance cost, performance, and flexibility within an SOC design framework.

The foundations for a processor-centric design flow are:

- Work top-down from the system's essential I/O interfaces and computation requirements.
- Use firmware-programmable processors pervasively to implement tasks because programmability adds low-cost flexibility to the system design while maintaining high performance levels.
- When tasks have specific computational patterns, optimize the processor to fit these tasks to minimize power and energy consumption and to maximize performance.
- When a task exceeds the capacity of an optimized processor, parallelize the task across multiple processors to keep clock rates low, which in turn will keep power and energy consumption under control.
- When a group of tasks fits together within a processor's capacity limit, map the tasks together onto one processor to minimize hardware cost, power, and communications overhead.
- Measure the communications traffic patterns through system simulation and optimize the software and hardware interconnects around those patterns.
- Start with early, rough simulation of communication tasks and refine the system into detailed implementations of processors, software, and other hardware accelerators, all running in increasingly accurate simulations.

The Starting Point: Essential Interfaces and Computation

The first step when designing a system is to identify the chip's essential I/O interfaces and the required computation or data-processing needs. The target product's marketing requirements usually establish the mandatory physical interfaces and necessary functions. These mandatory elements form the starting point for all other decisions—implementation decisions about these functions and decisions about the inclusion of additional supporting functions.

Not all interfaces and computation tasks are equally fundamental to the design of the system. For example, in an integrated disk-drive controller chip, the external interface to the read-head and the servo motor are essential to the chip's function, but external interface to buffer memory is not. Buffer memory could be implemented either on or off chip. That design decision requires more detailed analysis of cost and bandwidth tradeoffs. Similarly, a Secure Socket Layer security chip effectively requires implementation of the RSA (Rivest-Shamir-Adelman) algorithm for public/private key encryption, but the same chip may or may not include other TCP/IP protocol-processing functions.

How to parallelize a task

When a single task presents a particularly high computational load, designers must either choose a very fast general-purpose processor or pursue a more parallel hardware implementation. Fast general-purpose processors run at very high clock rates—for example, greater than 1 GHz. Processors that run at high clock rates are likely to be both unacceptably power-hungry and difficult to design and integrate into an ASIC or SOC without specialized skills in high-speed, on-chip design. For most embedded applications, an approach using parallel hardware resources is far more likely to fit the needs of an ASIC or SOC design project.

Historically, hand-coded RTL accelerators were the only viable choice for implementing parallel task execution but this design approach typically limited the complexity of the algorithms that could be implemented. Customizable microprocessor cores add a simple, efficient means to exploit task parallelism, especially fine-grained (instruction-level) parallelism and data parallelism (through SIMD or *single-instruction, multiple data* instructions).

Basic algorithm analysis and instruction-design process can be complex but it is also systematic. Automated, compiler-like tools can analyze a source-level program and develop task-specific instruction sets that will boost a processor's performance for the target algorithm, often by as much as an order or magnitude or more. Today's more advanced compiler algorithms—used for code selection, software pipelining, register allocation, and long-instruction-word operation scheduling—can also be applied to the discovery and implementation of custom instructions and code generation using those new instructions.

Automatic processor generation builds on the basic flow for application-specific processor generation but adds the creation of automatic processor architecture, as shown in
Figure **1**.

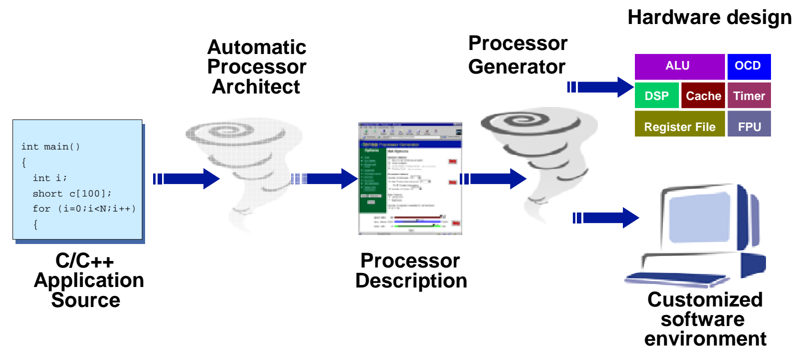

Figure 1: Automatic Processor Generation

Automatic processor generation offers many benefits beyond simple discovery of improved architectures. Compared to hand-crafted instruction-customization methods, automated tools eliminate the need to manually incorporate new data types and intrinsic functions into the application source code. These tools can accelerate applications that are too large or complex for a human programmer to assess and they can perform this analysis very quickly—often in as little as an hour. (The return on the time invested in automated exploration can be tremendous.) As a result, this technology holds tremendous promise for transforming the development of processor architectures for embedded applications.

The essential goals for automated processor generation are:

- A software developer of average experience should be able to easily use the tool and achieve consistently good results.
- No source-code modification should be required to use the automatically generated instruction sets. (Note that some ways of expressing algorithms are better than others for exposing the latent task parallelism, especially for SIMD optimization, so source code tuning can help in some cases. Any automated processor generator should highlight opportunities for source code improvement.)
- The generated custom instructions should be sufficiently general-purpose and robust so that any subsequent small changes to the application code do not degrade application performance.
- The design-automation environment should provide guidance so that advanced developers can further enhance the automatically generated custom instructions to achieve even better performance.

- The development tool must be sufficiently fast so that the design team can assess a number of potential extensions to the instruction set. Ideally, the tool should be able to evaluate thousands of architectures per minute.

Requirement for generality and in-system reprogrammability mandate two related use models for any automated processor-generation system:

- Initial SOC development: C/C++ in, instruction-set description out
- Software development for an existing SOC: C/C++ and generated instruction-set description in, binary code out

The XPRES (Xtensa Processor Extension System) compiler is an automated instruction-set generator for Tensilica's Xtensa customizable processor architecture. Figure 2 shows the four-step process for using the XPRES compiler. All of these steps are machine-automated, except for optional manual steps as noted.

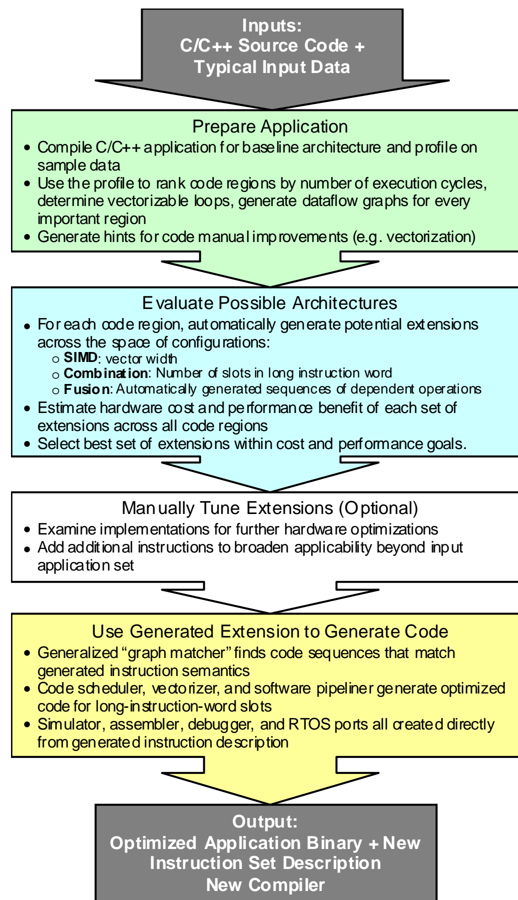

Figure 2: XPRES Automatic Processor Generation Flow

Automatic generation of a tailored C/C++ compiler adds significantly to the usefulness of an automatically generated processor. As the source application evolves, which it surely will, the generated compiler aggressively searches for opportunities to use the processor's custom instructions to streamline code and boost execution performance.

Automatic processor generation can even be an effective tool for generating fairly general-purpose architectures. So long as the basic set of operations is appropriate to another application, even if that application is unrelated to the first, the generated compiler will often use the extended architecture effectively.

The XPRES automatic processor generator internally enumerates the estimated hardware cost and application performance benefit of each of thousands of configurations, effectively building a pareto curve, such as that shown in

Figure **3**. Each point on the curve represents the best performance level achieved at each level of added gate count. This image is a screen capture from Tensilica's Xplorer development environment, for XPRES results on a simple video motion-estimation routine (sum-of-absolute-differences).

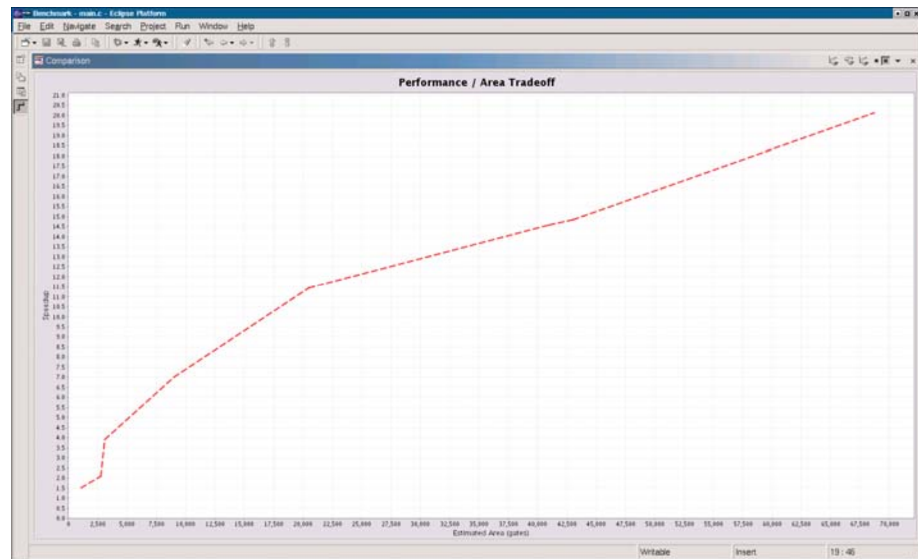

Figure 3: Automatic Generation of Architectures for Sum-of-Absolute-Differences

System designers can apply automatic generation of custom processor instructions to a very wide range of potential problems. The technique yields the most dramatic benefits for data-intensive tasks where much of the processor-execution time is spent in a few hot spots and where SIMD, wide-instruction (with multiple independent

operations per instruction), and operation-fusion techniques can sharply reduce the number of instructions per loop iteration. Reducing the number of instructions in a critical loop boosts the processor's ability to execute tasks in a minimal number of clock cycles, which reduces the need for high clock rates. Reducing the need for high clock rates helps to minimize a system's power consumption.

Media- and signal-processing tasks often fall squarely in the sweet spot of automatic architecture generation. The automated processor generator can also handle applications where the developer has already identified key task-specific functions. Table 1 shows the results of automated processor generation for three applications using the XPRES compiler, including one fairly large application: an MPEG4 video encoder.

Application	Speed Increase	Baseline Code Size	Code Size with Acceleration	MIPS32 Code Size (gcc –O2)	Configurations Evaluated	Generator Run Time (minutes)
MPEG4 Encoder	3.0x	111KB	136KB	356KB	**1,830,796**	30
Radix-4 FFT	10.6x	1.5KB	3.6KB	4.4KB	**175,796**	3
GSM Encoder	3.9x	17KB	20KB	38KB	**576,722**	15
GSM Encoder (FFT ISA)	1.8x	17KB	19KB	38KB	-	-

Table 1: Automatic Processor Generation Results for DSP and Media Applications

Table 1 includes code-size results for the baseline Xtensa processor architecture and for the customized Xtensa processor core automatically generated for each application. Using aggressively customized instruction sets can increase code size slightly, but in all cases, the optimized code remains significantly smaller than that for conventional 32-bit RISC architectures because the Xtensa processor architecture has a native 16/24-bit instruction set, which results in significantly smaller compiler-generated code as shown in the table comparisons with the MIPS32 instruction set. Table 1 also shows the number of configurations evaluated by the XPRES compiler, which increases with the size of the application. The XPRES compiler's run time also increases along with the size of the application, but averages about 50,000 evaluated configurations per minute on a 2GHz PC running Linux.

Table 1 also shows one example of generated-architecture generality. The GSM Encoder application source code was compiled and run on a processor that was optimized for the FFT application, not on a processor that had been customized for the GSM Encoder. While both applications are DSP-like, they share no source code. Nevertheless, the automatically generated C compiler for the FFT-optimized processor could recognize ways to use the processor's FFT-optimized instruction set to accelerate the GSM Encoder by a factor of 1.8 when compared to the performance of code compiled for the baseline Xtensa processor instruction set.

Completely automated instruction-set customization carries two important caveats:

1. Programmers often know certain facts about the behavior of their application, but these facts are not made explicit in the C or C++ code. For example, the programmer might know that a variable can only take on a certain range of values or that two indirectly referenced data structures can never overlap. The absence of that information in the source code will inhibit certain automatic optimizations in the machine code and instruction customization. Guidelines for using an automated instruction-set generator should give useful hints on how to better incorporate such application-specific information into the source code. A human creator of custom instructions may know this information and be able to exploit this additional information to create more efficient custom instructions.

2. Expert system architects and software developers can sometimes develop dramatically different and novel alternatives for task algorithms. A different inner-loop algorithm may be much better suited for custom-instruction acceleration than the original algorithm captured in the C or C++ source code. Very probably, there will always be a class of problems where the expert human will outperform an automated instruction generator, although the human will always take longer (sometimes much longer) to develop an optimized architecture.

Conclusion

The implications of automated instruction-set generation are wide-ranging. First, this technology opens up the creation of application-specific processors to a very broad range of ASIC and SOC designers. It is not even necessary to have a basic understanding of processor design or instruction-set architectures. The basic skill to run a language compiler is sufficient to use an automated tool for creating custom instructions.

Second, automated instruction-set generation is very good at dealing with complex problems where the application's performance bottleneck is spread across many loops or code sections. An automated, compiler-based method is easily able to track the opportunities to share instructions among loops, the relative importance of different code sections based on dynamic execution profiles, and the cumulative hardware cost estimate. Global optimization is far more difficult for a human designer to track.

Third, automated instruction-set generation ensures that custom instruction can be used by multiple applications without source-code modification. A compiler-based tool knows exactly what combination of primitive C operations corresponds to each new instruction, so it is able to instantiate that new instruction wherever it benefits performance or code density. Moreover, once the instruction set is frozen and the ASIC or SOC is built, the resulting C compiler retains knowledge of the correspondence between the C source code and the instructions. The compiler can use these same custom instructions even as the C source is changed.

Fourth, the automated processor generator can make better custom instructions than human architects in many cases. The generator is not affected by a human architect's prejudice against creating new instructions (design inertia) or influenced by architectural folklore about the rumored benefits of certain instructions. The automated tool has complete and accurate estimates of gate count and execution cycles. It can perform a comprehensive, systematic, and objective cost/benefit analysis. This combination of benefits therefore fulfills both of the key promises of application-specific processor cores for ASIC and SOC design:

- Less expensive and more rapid development of optimized chips
- Easier chip reprogramming to accommodate evolving system requirements after the chip has been fabricated

Note: If you would like to explore the power of automated processor generation for your next ASIC and SOC designs, contact Tensilica for a consultation.

US Sales Offices:

Santa Clara, CA office:
3255-6 Scott Blvd.
Santa Clara, CA 95054
Tel: 408-986-8000
Fax: 408-986-8919

San Diego, CA office:
1902 Wright Place, Suite 200
Carlsbad, CA 92008
Tel: 760-918-5654
Fax: 760-918-5505

Boston, MA office:
25 Mall Road, Suite 300
Burlington, MA 01803
Tel: 781-238-6702 x8352
Fax: 781-820-7128

International Sales Offices:

Yokohama office (Japan):
Xte Shin-Yokohama Building 2F
3-12-4, Shin-Yokohama, Kohoku-ku,
Yokohama
222-0033, Japan
Tel: 045-477-3373 (+81-45-477-3373)
Fax: 045-477-3375 (+81-45-477-3375)

UK office (Europe HQ):
Asmec Centre
Eagle House
The Ring
Bracknell
Berkshire
RG12 1HB
Tel : +44 1344 38 20 41
Fax : +44 1344 30 31 92

Israel:
Amos Technologies
Moshe Stein
moshe@amost.co.il

Beijing office (China HQ):
Room 1109, B Building, Bo Tai Guo Ji,
122th Building of Nan Hu Dong Yuan, Wang Jing,
Chao Yang District, Beijing, PRC
Postcode: 100102
Tel: (86)-10-84714323
Fax: (86)-10-84724103

Taiwan office:
7F-6, No. 16, JiHe Road, ShihLin Dist,
Taipei 111, Taiwan ROC
Tel: 886-2-2772-2269
Fax: 886-2-66104328

Seoul, Korea office:
27th FL., Korea World Trade Center,
159-1, Samsung-dong, Kangnam-gu,
Seoul 135-729, Korea
Tel: 82-2-6007-2745
Fax: 82-2-6007-2746