Streaming and Dynamic Compilers for High Performance Embedded Computing

Dr. Peter Mattson (Mattson@reservoir.com), Senior Engineer
Dr. Jonathan Springer (Springer@reservoir.com), Senior Engineer
Mr. Charles Garrett (Garret@reservoir.com), Senior Engineer
Dr. Richard A. Lethin (Lethin@reservoir.com), President
Reservoir Labs, Inc.
628 Broadway, Suite 502
New York, New York 10012
Phone: (212) 780-0527 – Fax: (212) 780-0542

Abstract:

It is possible to design programmable embedded processors with peak computational rates and area/power efficiencies near that of fixed function hardware, but compiler technology is critical to programming these processors effectively. This talk will cover two programming and compiler technologies among those proposed for addressing the challenges of these processors: streaming languages and compilers, and dynamic compilers. It will present the hardware trends that motivate streaming languages, discuss the semantics of streaming languages, and describe emerging streaming compiler techniques such as those used by Reservoir's R-Stream™ compiler. Also, this talk will explore the benefits of dynamic compilation in the context of reconfigurable computing, using Reservoir's R-DYN™ compilers for illustration.

Two architectural trends are driving next generation high-performance computing. The first trend is toward multiple processing elements on a single chip. These elements tend to be computationally powerful (multiple functional units), with minimal control overhead (VLIW, possibly SIMD). The second trend is toward explicit data transfers between processing elements and/or memory (e.g. DMA engines, on-chip networks). Multiple processing elements demand abundant parallelism; explicit data transfers require understanding the high-level data-flow of the application. Meeting these demands requires the development of more powerful compiler techniques to extract parallelism and data-flow information, new programming languages and APIs that make these characteristics explicit, or some combination of the two.

Stream programming languages make high-level data-flow explicit and enforce data parallelism. They compose an application from a series of kernels applied to sequences of data records called streams. A kernel can be thought of as a restricted loop that iterates over the records in one or more input streams and produces the records in one or more output streams. The streams used as kernel inputs and outputs explicitly describe the high-level data-flow of the application. The kernels enforce data-parallel processing of records within a stream.

This talk will explore several key semantic design decisions of streaming languages and their implications for usability and compilation. One such decision is how to describe a streaming application. One approach is to use a data-flow graph composed of kernels connected by streams. The opposite approach is to use an imperative program that calls kernels like functions with streams as arguments. Another important decision is how strongly a kernel enforces parallelism. Certain minimum restrictions, such as disallowing access to global data within kernels, are common to most streaming languages. Further restrictions represent tradeoffs between expressiveness and parallelism, or between different compiler analysis requirements.

Streaming languages provide the compiler with a consistent, analyzable structure, enabling new, powerful compiler techniques for mapping that structure to high-performance hardware. For instance, a streaming compiler can use this structure to load balance across multiple processors by dividing, combining, or replicating kernels, or manage all on-chip memory at compile time based on the data-flow of streams between kernels.

Streaming languages can yield impressive performance when executed by processors designed to take advantage of the structure they provide. StreamC/KernelC, a streaming language developed at Stanford, has been used to develop complex applications such as stereo depth extraction and MPEG encoding for the

Imagine Media Processor that sustain 10-20 GOPS on a single chip. Reservoir's R-Stream™ compiler builds on the compiler technology developed for StreamC/KernelC.

This talk will also cover dynamic compilation, the run-time compilation of application code, which has recently emerged as a versatile technique for improving performance and managing difficult issues such as cross-platform portability. Dynamic compilation offers several advantages over ahead-of-time compilation:

- Platform-specific optimization. Programs compiled ahead-of-time are typically designed to be compatible with a family of processors. Dynamic compilation can take advantage of the special features of a specific instance of that family that is used for a given run.

- Profile-guided optimization. Dynamic compilation can analyze the run-time behavior of a program for valuable clues to optimization. This information is typically difficult to obtain for ahead-of-time compilation. When it is available, it often reflects a specific "training run," which may differ from the behavior of actual use.

- Cross-module optimization. It is difficult for ahead-of-time compilers to optimize across distinct compilation units, or into libraries, particularly dynamically-loaded libraries. At runtime, however, all libraries are known.

- Abstraction for software engineering advantages. Dynamic compilation makes it practical to have a layer of abstraction separating program from platform. This abstraction layer creates engineering benefits for portability and maintainability.

There are a variety of examples of dynamic compilation, both in the research world and in production systems. Research efforts include IBM's Daisy project, HP's Dynamo, rePLay from the University of Illinois, and DyC from the University of Washington. Examples of production use include Digital FX!32, Transmeta Crusoe, Sun's Java JDK, Microsoft .NET, and Reservoir's R-DYN™.

Dynamic compilation is well within the reach of current desktop processing power, and can now filter down to the embedded space. Reservoir's R-DYN™ Java dynamic compiler spends a few milliseconds per procedure in full-optimization mode. Lightweight compilation is roughly an order of magnitude faster (with a tradeoff in performance).

This talk will explore the potential of dynamic compilers on reconfigurable embedded platforms. Dynamic compilation is a natural counterpart to reconfigurable computing, because platform-dependent optimizations can change with the architecture. Such optimizations include:

- Cache-sensitive optimization. Inlining and loop unrolling are very sensitive to the size of the instruction cache, and available memory. Heap object alignment and loop tiling optimizations depend on data cache size and shape.

- Register pressure optimization. Register allocation, memory promotion, and scalar replacement are compiler transformations that depend on the characteristics of the register file.

- Instruction scheduling. Changes in the organization or availability of functional units within a CPU require corresponding changes to the instruction order to maximize performance (and on some architectures, to obtain correct behavior).

Reliability of dynamic code generation can be managed through a variety of techniques, beginning with aggressive testing of expected as well as randomly-generated source. If desired, compilation can be limited to procedures that have been tested ahead-of-time; Reservoir has applied this technique successfully in mainframe emulation on stock hardware. Furthermore, use of a dynamic compiler can increase the robustness of the overall system, by validating code and recompiling to work around hardware failures.