How Code Generation Will Save Moore's Law

William I. Lundgren (wlundgren@gedae.com),
Kerry B. Barnes (kbarnes@gedae.com), James W. Steed (jsteed@gedae.com)
Gedae, Inc., 1247 Church Road, Suite 5, Moorestown, NJ 08057

Introduction

If software is to save Moore's Law, advanced programming tools must be in place to save software development. Chip density will eventually reach a limit where a single chip cannot be made any faster. The choice must be made (and is being made today) to spend money adding more hardware components rather than enhancing single processors. These multi-component systems rely on massive parallelism and pipelined processing to continue the trend of Moore's Law in creating faster applications and doing more operations in real-time. However, more complex systems are also much more difficult to program. Such systems are likely a heterogeneous collection of processors combining lightweight processors (AltiVecs, TigerSHARCs) with hardware components (FPGAs, ASICs), attempting to fit the most processing power possible into a single board or chassis. Teams of scientists with diverse, specialized knowledge must be assembled in order to develop an application on these hardware systems, resulting in long development times and prohibitive costs for development of new software.

Parallelism and Distribution in Software Development

In order to support programming modern hardware systems, parallelism and distribution must be integral parts of software development toolkits. The toolkits must be able to support a wide range of processors, from workstations to DSPs to FPGAs, and create code which implements the entire system, from the firmware which does the front end processing to the GUI that runs on the workstation to analyze the data, including all interprocessor communication. If the development of hardware systems with more programmable components is to fuel Moore's Law, this evolution is of no practical use if the components are not easily programmable and if their interaction is not addressed by the toolkits.

Furthermore, these toolkits must allow the distribution of processing to be easily reconfigured, both for experimentation during development and for easily adapting to future generations of hardware. Currently many software development projects involve re-implementing the same algorithm on a new system. Even if the system comes from the same vendor with the same board support package interface, the processing must be redistributed in order to achieve the higher throughput provided by the increased number of processors or their higher speeds. However, under many software development methods, the parallelism of the legacy software is explicit in the code, and the development team must make the choice to analyze the code to find and alter the parallelism or start from scratch. Certainly algorithm development cannot help fuel the future development of Moore's Law if so many algorithm developers must focus on the rote work of re-implementing yesterday's ideas.

Software Obsolescence

Software development tools must maintain a separation between the functional description (what data processing is to be done) and the implementation detail (target-specific information necessary for implementation, such as the distribution) in order to help maintain the course of Moore's Law and easily adapt to advances in hardware. The code generation used by these tools should create another level of abstraction over source code. Some of the implementation detail that is included in source code should not be necessary when applying code generation. While a graphical representation is used in place of source code by many code generators, the implementation detail (such as sends and receives) is still included in the graphical representation. In such cases, the code generator has added little value. The application is not portable and cannot be easily reconfigured. When the next generation of hardware arrives, the application must still be analyzed to determine the parallelism or, failing that, the development team must fully re-implement the application. Improvements in coding productivity are also limited because this implementation detail is such an integral part of the software design. The essential functionality of the application is obscured by the amount of implementation detail added to the specification, making the application much more difficult to program than the algorithms that form it. This separation between functionality and implementation can be done by maintaining two

sources of application information – one that describes the functionality and a second that specifies how that functionality is to be implemented. Many copies of the implementation specification can be maintained for a single functional description.

Challenges for the Code Generator

In order for the code generator to successfully create this separation, there are two key challenges. The first challenge is to discover the essential functional information that must be included in the programming language. Functionality specified by the programming language must describe the application sufficiently such that when the code generation makes radical changes to the implementation, the specified behavior is maintained. A second challenge is the need for intelligent algorithms that transform the functional and implementation specification into an implementation that runs on a heterogeneous (RISC, DSP, FPGA, and other architectures) multiprocessor target. Many concepts have been used in developing programming languages for code generation, however few tools have adequately addressed these two challenges to allow a wide range of applications to be developed for a wide range of heterogeneous targets. Data flow is a good abstraction that meets the needs of some functional requirements. Object oriented concepts provide for the localization of related functionality. State diagrams can directly express part of the functionality. Each is limited. We need a fully integrated language to directly state the diverse functional requirements of software development,

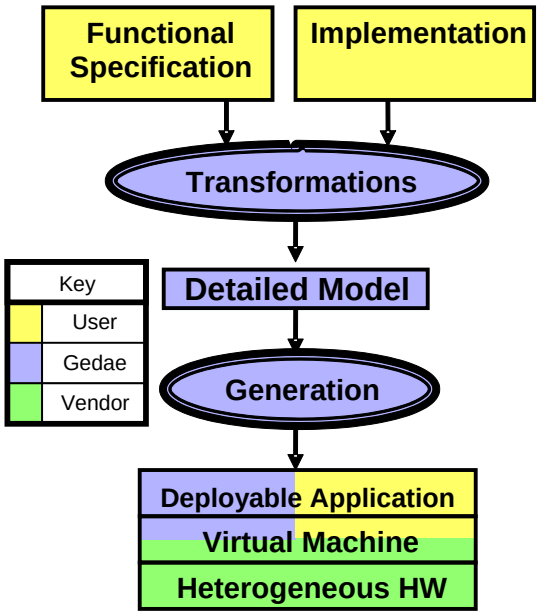

Figure 1 – Structure of Gedae

extending each idea as necessary. As the language is being developed we must also create a full suite of algorithms that transform the functional description into an implementation. The suite must take into account the variability of the implementation specification and target, and it must not sacrifice efficiency.

Gedae is an example of such an approach. The structure of Gedae is shown in figure 1. This paper will discuss how Gedae has been designed to support the development of software for next generation hardware systems which combine more and more components on a smaller footprint. It will discuss how Gedae's language has been designed to allow a wide range of applications to be developed, it will highlight several of the 100+ transformations Gedae uses to modify the user's specification into an efficient implementation, and it will describe Gedae's virtual machine which enables wide portability, as well as recent extensions to the virtual machine which support targeting FPGAs. Figure 2 shows the structure of Gedae adapted to program hardware architectures – including FPGAs and arrays of processors with embedded memory. As the parallelism of hardware increases to keep pace with Moore's Law, tools like Gedae will become more and more necessary for dealing with the complexities of multiprocessor implementations and avoiding the pitfalls of software obsolescence.

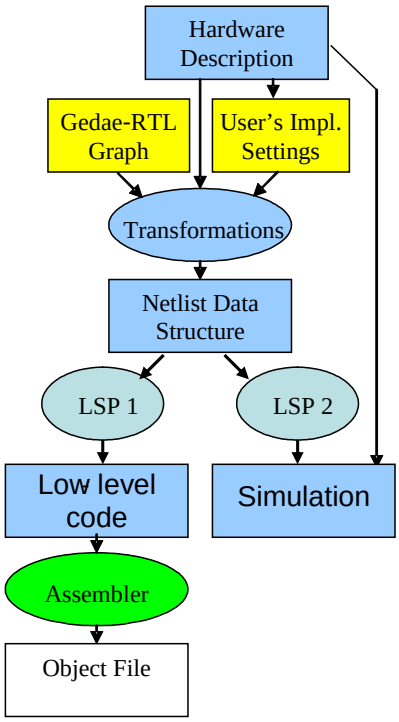

Figure 1 - Structure of Gedae Adapted to Program Hardware for example FPGAs or PIMs