Combating obsolescence in high-performance multiprocessor software

By William Lundgren, Kerry Barnes, and James Steed

When software is customized to its hardware, it is difficult to break that bond. Software is then at risk for becoming obsolete just like the hardware. Three methods for combating this obsolescence are presented: specification documents, middleware, and model-driven development. Only model-driven development is capable of addressing this issue while simplifying software development, making global-level optimizations to the application's structure and providing a general purpose solution instead of a finite set of algorithms.

One doesn't think of software becoming obsolete in the same way that hardware becomes obsolete. With hardware, once the circuit is no longer fast enough or no longer suited to the desired application, it has to be thrown away. Meanwhile, software can be rewritten. On the surface, rewriting software seems like a cheap and simple task. However, usually when software becomes obsolete, it is not just a matter of changing a few lines of code.

Take a hardware refresh many engineering teams may be considering right now: moving a deployed application from a quad DSP board (for example, PowerPCs or TigerSHARCs) to a multicore architecture like the IBM/Sony/Toshiba Cell Broadband Engine (Cell BE). To write a real-time program on four DSPs, the development team must make many architecture-specific optimizations. First, they must optimize compute-intensive subroutines to execute efficiently on a single DSP, for example vectorizing the arithmetic in the code and observing byte alignment to best utilize the four ALU data paths on the PowerPC processor. Portions of the code might even be written in Assembler instead of C or Fortran in order to milk each ounce of performance possible. Second, they must partition the code to run on four processors and coordinate data movement and sharing. Porting this type of DSP application with all its architecture-specific optimizations to a processor with a completely different architecture and completely different optimizing strategies is no easy task. If a new development team is handed the DSP code and asked to retarget it to the Cell, divining the algorithmic IP from the current DSP code is likely impractical and may even be impossible. This type of software retargeting likely involves beginning at square one: algorithm development.

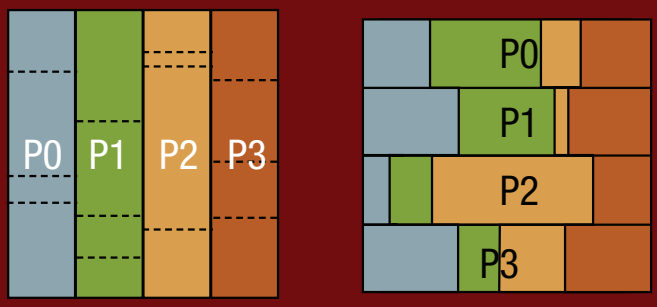
Several software development methodologies have been developed to combat this type of software obsolescence, though, including software spec documents, middleware, and model-

The challenge of multiprocessor software portability

If software is portable, it can be run on multiple types of hardware. Software portability reduces the risk of obsolescence if that portability includes future types of hardware. To study the challenge of software portability, let's look at a simple example: a distributed sort application. Assume that the data set is very large and at the beginning of the sort, each processor has N/P data elements where N and P are the data size and number of processors. There are two sections to this algorithm: First, the processors must decide together how to redistribute the data, then they must partition and redistribute their arrays accordingly.

In the first section of this algorithm, each processor locally calculates a histogram, and the histograms are collected to determine how the data should be redistributed. To perform this communication, an adder tree is formed where sibling processors combine their histogram with their neighbor and send the result up the tree's hierarchy. After the histograms are combined and collected, they are broadcast to all the processors.

In the algorithm's second section, each processor partitions its array and distributes the data to the other processors. This communication is very similar to a corner turn (a distributed matrix transpose) as each processor is sending and receiving part of its row of data to every other processor. An important difference is that the matrix rows in this transpose are broken into segments of varying lengths. At most, N/P data elements can be included in each transfer, but the actual number of elements transferred is usually much less than N/P. An illustration of this variable-width corner turn is shown in the figure.



To perform this algorithm, three communication protocols are used: a collect, a broadcast, and a variable-width corner turn. This communication is central to the algorithm; the only other functions used in the algorithm are calculating the histogram, partitioning, and sorting.

driven development. We will present an example application and explore how these three methods for combating software obsolescence address porting the application to a new architecture.

Documenting the software

One approach to battling obsolescence is through the use of detailed specifications. Commonly, software development teams require the algorithmic IP be specified in a detailed document and then insist the implementation exactly follow the specification document. This approach prevents the new development team from having to divine the meaning of legacy code. However, it does not serve the primary objective of reducing cost; the development team still is left with the task of rewriting the code, in its entirety, for the new processor.

For the distributed sort application (see sidebar, page 34), a document could be created that describes the distributed histogram, the variable-width corner turn, and all the other functions in the algorithm. To implement a distributed sort on new hardware, the engineer creates the software that realizes each of these specifications.

This solution relies on the engineer to deal with much of the complexity of implementing distributed software. The programmer must know how to best utilize the parallel processing paths and communications bandwidth of the multiprocessor system. This knowledge is very specialized; a programmer with expertise in programming one DSP system from one vendor does not necessarily have equivalent knowledge for a DSP system from another vendor, let alone a multicore processor with radically different hardware.

Corralling the software architecture in middleware

Another methodology for dealing with obsolescence is to use middleware to separate the architecture-specific code from the algorithmic IP using a layer of software. This approach addresses many issues in software portability. However, if the task at hand is moving from a board of four DSPs, where each processor has large memories, to eight lightweight processing cores with tight memory restrictions on both program size and data size, then middleware offers no assistance. The software architecture – how the software is distributed across the processing elements – must be changed to meet the architecture of the hardware system, and middleware cannot abstract away the software architecture.

Middleware addresses the portability of the application by providing a platform-independent method of specifying common operations. For example, functions for common routines like local sorting or broadcast are provided out of the box, and if code is written using these functions, the code is portable. Less common functions are not supported.

Middleware likely doesn't support the variable-width corner turn from the distributed sort algorithm. The standard corner turn (with fixed-width rows) might be supported, but data must be padded to use the routine, losing efficiency. The routine could be hand-coded, but this coding is difficult and likely presents a trade-off between efficiency and portability. Also, if hand-coding is chosen, while the corner turn from a middleware library might provide portability from four processors to eight, foresight is required to parameterize the parallelism of the system into the hand-coded routine.

Similarly, a histogram adder tree is not likely to be supported. The adder tree can be hand-coded, combining the calls to the middleware communications library with calls to the middleware vector arithmetic library. However, this combination is inserting the software architecture into the code. If the number of processors increases, the code must be reviewed and rewritten to include the additional levels to the tree. If the interconnection of the architecture changes from a token ring to a bus or other connectivity, the tree structure might change according to which processors are now nearest neighbors in the new interconnection.

As software gets more complex, the struggle to keep the software architecture out of the algorithms gets harder. Consider a complete radar application with multiple modes of operation, such as searching for targets and tracking them once they are found. It is not just one algorithm like sorting, but multiple algorithms running at the same time, with each one requiring a different number of processors and different distribution across those processors. In a real-world application, the impact of adding the software architecture to the code is essentially irreversible.

Automating the software architecture with model-driven development

The third approach to thwarting software obsolescence issues is that of model-driven development. This approach captures the algorithmic IP in a functional model that is very much like an executable version of a specification document. The toolset includes a multiprocessor compiler, which uses its global view of the application along with its knowledge of the target hardware to transform the functional model into a target-optimized implementation.

Model-driven development accomplishes its mission by incorporating the software architecture during compilation instead of directly inserting it into the code. The removal of the software architecture from the code increases portability, allowing one functional model to be used on any multi-CPU system, regardless of processor type, interconnect, or other fine details of the hardware. The compiler has a unique global view of the application and is able to use that information to automate and optimize much of the application's structure – optimizations that are difficult when hand-coding, even with middleware. Let's consider how Gedae, a programming language and multiprocessor compiler, assists in implementing the same sorting algorithm.

Gedae's compilation process and how it affects software development teams is shown in Figure 1. Application algorithms are captured in a functional model. To form the application from the model, the Gedae compiler first turns groups of algorithms

into threads. Forming threads is an essential part of parallelizing code, and this type of compiler is able to do it automatically by analyzing the relationships between the data streams in the algorithms to plan the structure of the threads. In the distributed sort algorithm, the corner turn presents a natural boundary to separate the application into threads that can be processed concurrently.

To implement this application for an architecture like the Cell BE, the developer must use memory very efficiently. Model-driven development tools use their global view of the application to analyze the size and time of use of each buffer. Using this information, the compiler is able to preplan memory usage so it can be statically allocated and shrink the total memory footprint by reusing buffers as much as possible.

Next, the compiler automatically inserts and manages all communication needed to sort the data. While this variable-width corner turn can be seen as a gap in the middleware library, a model-driven development tool is able to automatically generate an efficient implementation. Gedae's graphical language breaks processing into components, and any connection between components – even if those components are in the same thread – can be turned into a transfer between processors in the generated application. Figure 2 shows the portion of the flow graph from which the corner turn is created during compilation. There are P instances of the `vui_part_vvui` (partition) and `vvui_nconcat_vui` (concatenate) components, one for each processor. The connection between the two sets of components creates the (p,q)-to-(q,p) pattern necessary for a corner turn, allowing the compiler to automate the rest of the implementation. Using its global view of the application, model-driven development is also able to provide a rich set of debugging and analysis tools. For example, the Gedae Trace Table in Figure 3 shows all timing information including communication [green (send) and red (receive) bars].

In this example, Gedae has minimized the work required to implement the intricate communication patterns. The compiler has automatically created P sets of threads to distribute to the P processors and will automatically manage the concurrency of the threads. Because Gedae is managing the distribution of the application at build time, the functional model loses no portability, and the software architecture is not inserted into the functional model. Software development's complexity is mitigating without losing any performance, and the same software can easily be moved from one system to another, whether it's a quad DSP board, the Cell BE, or a future architecture.

Model-driven development future-proofs software

The rise of multicore processors like the Cell BE and the Intel Core 2 Duo/Extreme increases the challenge of software portability and will cause more software products to be at risk for obsolescence. Dealing with different multiprocessor systems is no longer limited to massive software projects built for embedded hardware or supercomputers; all

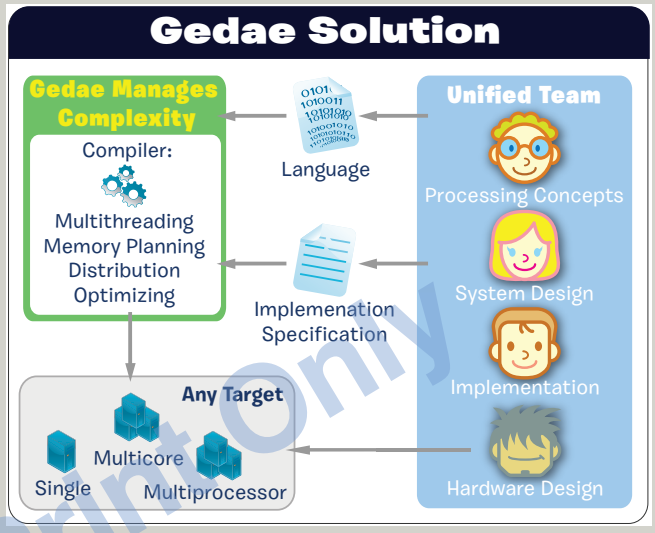


Figure 1

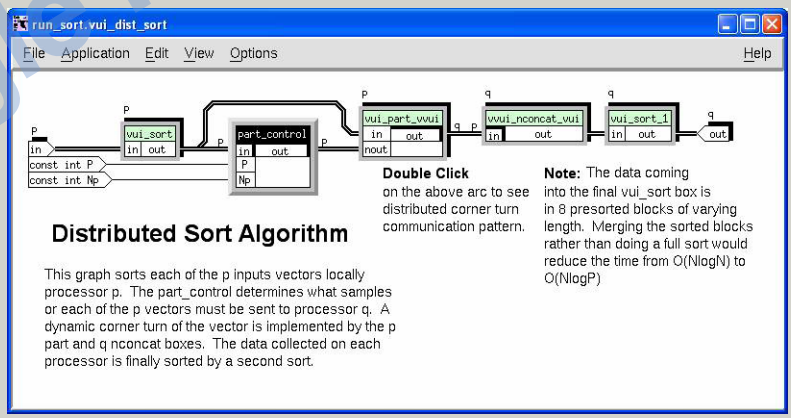


Figure 2

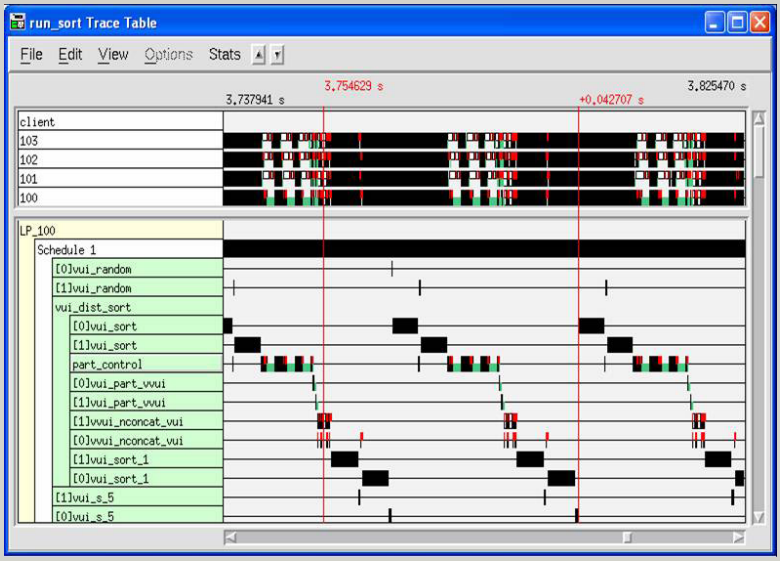


Figure 3

software must address these issues as we can no longer assume code will run on only one processing core. If software developers do not address these portability issues, they risk adding significant costs to maintain their software when their current chip is no longer fast enough. To avoid these issues, three methodologies have been developed: specifications documents, middleware libraries, and model-driven development. While all three methods help address the issues of porting software to new architectures, model-driven development tools like Gedae present unique advantages. These tools can both automate much of the work of creating the new implementation and use their global view of the application to make optimizations that are difficult when using the other two methods.

William Lundgren *is cofounder, president, and CEO of Gedae, Inc. William started his professional career at Corning Glass Works as a product development physicist. He later worked at the U.S. Air Force Institute of Technology and the U.S. Air Force Research Laboratories developing new speech and audio processing technologies. After leaving active duty in 1985, he moved to RCA Advanced Technology Laboratories, which became Lockheed Martin. He then spent the past 16 years leading Gedae development. William has a BS in Physics from Rensselaer Polytechnic University, BS and MS degrees in Electrical Engineering from the U.S. Air Force Institute of Technology, and is All But Dissertation for his PhD in Electrical Engineering from the University of Pennsylvania.*

Kerry Barnes *is chief scientist and a founding member of Gedae. Before joining Gedae, Kerry was a principal member of the engineering staff at Lockheed Martin, ATL where he was responsible for signal processing systems software/hardware, single-chip FFT design, direct digital frequency synthesizer design and implementation, and various software tools and applications development projects. He earned a BS in Electrical Engineering from Lehigh University and an MS in Computer and Information Science from the University of Pennsylvania.*

James Steed *is director of software development and a founding member of Gedae. Prior to Gedae, James worked at Lockheed Martin where he was responsible for developing the embeddable library of functions, including testing and creating a database and search utility. His most prominent project is the development of Gedae's new RTL language. James earned a BS in Computer Science from Cornell University and an MS in Computer Science from North Carolina State University.*

Gedae, Inc.
1247 North Church Street, Suite 5
Moorestown, NJ 08057
856-231-4458
wlundgren@gedae.com
kbarnes@gedae.com
jsteed@gedae.com
www.gedae.com