Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

Going Parallel With Prism

New Analysis Tool Helps Programmers Refactor Serial Code

By Tom R. Halfhill {4/27/09-01}

It'll be a long time—maybe forever—before someone invents a magic compiler that transforms existing serial code into optimized parallel code. Meanwhile, hard-pressed programmers are tackling the job by hand. Line by line, they pore over their programs, resolving data dependencies and rearranging functions for parallel execution on multicore processors. It's drudge work, but somebody has to do it. Lots of somebodies.

Now their task will be a little easier. CriticalBlue has introduced Prism, a code-analysis tool that helps developers extract thread-level and system-level parallelism from legacy programs written in sequential code. After running the target program in a software simulator that captures dynamic trace data, developers can use Prism to analyze the results in numerous ways. With raw statistics, expandable tables, telescoping charts, and data-flow graphs, Prism reveals detailed views of a program's mechanics. It can identify data dependencies that thwart parallelism, suggest modifications to source code, and estimate performance improvements.

Most important, Prism helps developers explore various what-if scenarios so they can make intelligent decisions *before* rewriting any code. Will modifying two or more functions to eliminate their interdependencies actually increase throughput? Will rewriting a different function yield a bigger payoff? Will a single-threaded program run twice as fast on a dual-core processor? Will the same program run four times faster on a quad-core processor? When does the overhead of creating and terminating threads outweigh the gains of greater parallelism? Without this foreknowledge, programmers can waste months on dead-end experiments.

Prism is a new direction for CriticalBlue, though it builds on the Scottish company's expertise. Until now, CriticalBlue's flagship product was Cascade, a design-automation tool. Cascade helps developers convert high-level software functions written in C or C++ into the RTL required to create hardware coprocessors. Prism is strictly a code-analysis tool, not a hardware-generation tool, but the experience gained from analyzing dynamic traces of program execution for Cascade is the foundation for Prism. Although Cascade remains a live product, the rise of multicore processors is creating demand for new parallel-programming solutions.

Like Cascade, Prism is for embedded-system developers, not programmers writing software for PCs and servers. It currently supports ARM and MIPS processors intended for multicore SoCs, but not ARC, x86, SPARC, or Tensilica processors. Nor does Prism support GPUs, an increasingly popular alternative for parallel processing. Versions for the Power Architecture and Renesas SH-Mobile architecture are in development. CriticalBlue says additional CPU architectures are relatively easy to support and will depend on customer demand.

Prism requires a separate simulator to gather the dynamic trace data, and it's limited to symmetric multiprocessing (SMP) on shared-memory systems. Currently, it doesn't support asymmetric multiprocessing (AMP)—a common system architecture in embedded systems using two or more processors. CriticalBlue says most customers wanted SMP support first.

Compatibility With Existing Tools

CriticalBlue delivers Prism as a plug-in module for the Eclipse Integrated Development Environment (IDE), an open-source

```
err = 0;
  for (r = 0, i = 0; r < mb_rows; ++r) {
    for (c = 0; c < mb_cols; ++c, ++i) {
      // pack each macroblock arguments and launch thread
      mbe_arg_pack(&arg[i], image, r, c, huffman, 0);
      // each mb created locally
      pthread_create(&thread[i], NULL, mbe_func, &arg[i]);
    }
  }

  // wait for threads to finish
  for (i = 0; i < mb_rows * mb_cols; ++i) {
    pthread_join(thread[i], (void *)&status);
    if (!err && arg[i].err) err = arg[i].err;
  }
```

Figure 1. Prism assumes most programmers will implement multithreading with the Posix Thread API. Although Prism can suggest places where threading will improve performance, it doesn't automatically generate parallel code or modify existing code. CriticalBlue says early customers were adamant about keeping their programs untouched.

framework rapidly gaining popularity among programmers. Eclipse isn't specific to any particular CPU architecture, operating system, or programming language. By delivering Prism as an Eclipse plug-in, CriticalBlue avoids the considerable effort required to develop a proprietary IDE and convince programmers that it's better than existing tools.

Nor has CriticalBlue reinvented the wheels of compilers or parallel programming. Prism works with any compiler and assumes the legacy code was written in ANSI-compliant C, C++, or assembly language. Currently, Prism assumes most programmers will implement thread-level parallelism using the Posix Threads library. Posix "pthreads" are a platform-agnostic way to implement multithreading in C and C++ (see Figure 1). For Toshiba, a lead customer, Prism also supports the Venezia "V-thread" API. CriticalBlue says Prism could support virtually any threading library, if customer demand warrants it. Although programmers who aren't using Eclipse and pthreads may still find Prism useful as a code-analysis tool, it works best for programmers who are coding in C, with Eclipse as their front-end IDE.

Using Prism is a five-step process. First, Prism analyzes the dynamic trace data gathered by running the target program in a software simulator. Next, before rewriting anything, programmers study the results and explore various alternatives for parallelizing the code. The third step is to choose the best alternatives (whatever the developer's priorities) and write the necessary code, starting with Prism's suggestions. Next, Prism checks for problems that could interfere with proper execution and optimum performance. Finally, programmers can repeat the process to make additional performance improvements.

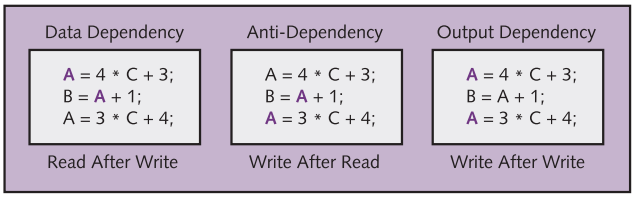

Figure 2. Prism analyzes dynamic trace information to identify three classic types of data dependencies that prevent certain operations from executing in parallel.

As mentioned above, Prism doesn't include a software simulator. Instead of reinventing that wheel, too, CriticalBlue assumes each developer will use a simulator already available for the target platform. For ARM processors, this might be the QEMU-based simulator in CodeSourcery's Sourcery G++. (QEMU is an open-source microprocessor emulator.) For MIPS processors, it will probably be MIPSsim, from MIPS Technologies. For Power Architecture processors, the likely choice will be Virtutech's Simics.

Running a program in a simulator can take many hours, depending on the program's size, its functions, the simulator's efficiency, and the speed of the computer on which the simulator runs. Simulation generates a prodigious amount of trace data that records every function call, memory transaction, and other operation the program performs. Data about memory references is particularly important, because it allows Prism to find the interdependencies that prevent some functions from executing in parallel.

Dependencies: The Enemy of Parallelism

Prism can identify all three types of classic data dependencies. One is a "true" data dependency, also known as read after write (RAW). Another is an antidependency, or write after read (WAR). The third is an output dependency, or write after write (WAW). Figure 2 illustrates some examples of arithmetic operations that can create these dependencies.

RAW dependencies can't be fixed, because they occur when an operation needs the result of a previous operation. For instance, a function that calculates the annual sales of a company by summing the monthly sales can't proceed until a previous function sums the weekly sales. The two operations have a true data dependency, so they can't execute out of order or in parallel. However, it still may be possible to improve a program's overall performance by rescheduling some of these RAW dependencies around other dependencies.

WAR antidependencies are relatively easy to fix. These may occur if the processor runs out of available registers to hold a working data set, or if two or more operations need to use the same memory, variable, or buffer. A good programmer or optimizing compiler can eliminate antidependencies by improving the allocation of these resources.

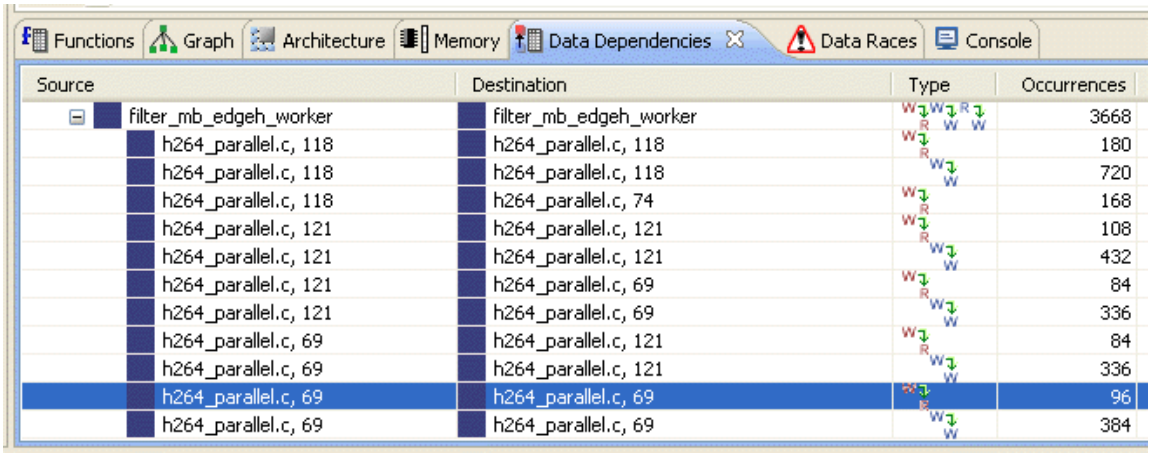

Figure 3. Finding data dependencies with Prism. Columns on the left identify the functions, source files, and lines of code in which Prism has found data dependencies. Columns on the right identify the types of dependencies (read after write, write after read, or write after write) and the number of occurrences that Prism found in the trace data after a simulation run. In this example, the programmer has highlighted a write-after-read (WAR) dependency in line 69 of the "h264_parallel.c" source file.

WAW output dependencies occur when an operation needs the result of a previous operation that hasn't yet saved its result in memory. If allowed to proceed, the second operation will read obsolete data. Some microprocessors have data-forwarding logic that automatically detects an instruction-level WAW dependency and forwards the pending result to a subsequent operation that needs it. Likewise, some processors with data caches can automatically intercept a read operation and redirect it to the pending data held in the cache. These techniques enable instruction-level parallelism and are usually transparent to programmers. To enable thread-level parallelism, however, programmers must fix WAW dependencies by rearranging some code.

Simply finding and identifying all these data dependencies in existing sequential code is half the battle. They can be very subtle. Often they aren't visible in neighboring lines of source code. Different functions scattered widely throughout a program may have mutual dependencies that aren't immediately apparent. Pointers and other indirect memory references can hide these hazards, too. Prism has several features that help programmers find, analyze, and fix these problems.

Rooting Out Dependencies

Figure 3 is a cropped screen photo of Prism analyzing a program that encodes and compresses video streams into H.264 format. Prism has found and identified thousands of

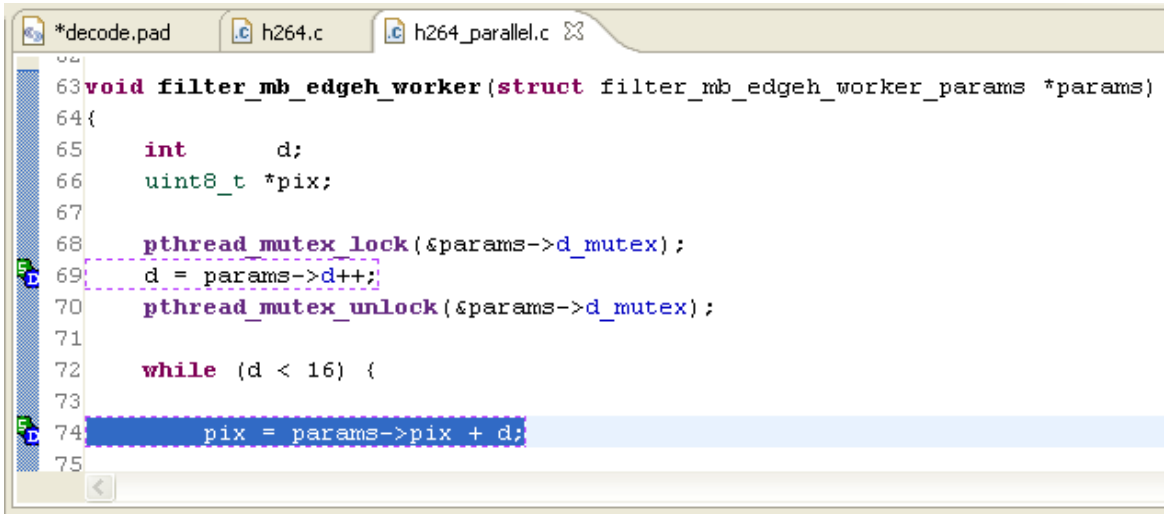

Figure 4. Analyzing source code with Prism. When the programmer clicks on a WAR dependency highlighted in Figure 3, Prism automatically loads the C source file ("h264_parallel.c"), scrolls the Eclipse text editor to the corresponding function (**filter_mb_edgeh_worker**), outlines the offending line of code (line 69), and indicates that this line is both the source and destination of the dependency. In this example, the function is performing two operations on the same variable in the same statement, potentially creating a WAR dependency.

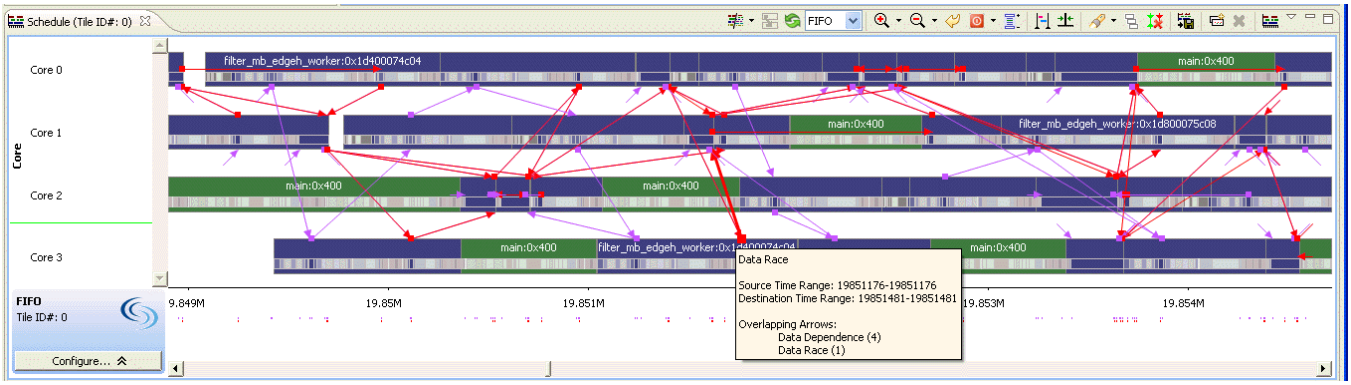

Figure 5. A multithreaded data-flow graph in Prism. In this example, four threads are running in parallel on a quad-core processor. Red arrows indicate potential data races between threads. Arrows pointing from left to right indicate the correct data flow—forward in time. Arrows pointing backward indicate problems—needed data won't be ready, causing a thread to stall until the data is available.

dependencies in the trace of an H.264 function (**filter_mb_edgeh_worker**) that performs edge detection on macroblocks (groups of pixels). This particular function suffers from all three types of dependencies. Fixing them won't require a programmer to rewrite thousands of lines of code, though. Only a few statements (indicated by their line numbers) are responsible, because the program calls this function often.

When the programmer selects a dependency in the table shown in Figure 3, Prism automatically displays the corresponding line of source code in the Eclipse text editor. In a narrow column to the left of the source code, Prism indicates whether that code is the source, the destination, or both the source and destination of the dependency. Figure 4 shows a cropped screen photo of the text editor.

Although some tasks in this example program are threaded already, they aren't properly synchronized, so they may create data races when executing in parallel—but not always. For programmers, this hazard is particularly maddening. A program can run properly on one type of multiprocessing system (say, a dual-core processor) but fail on another (such as a quad-core processor). Simply changing the degree of parallelism can alter the thread scheduling in ways that cause a function to stall while waiting for the results of another function. Sometimes, a program works correctly for years before the problem appears. (See *MPR 4/30/07-02*, "The Dread of Threads.")

Prism can reveal these hidden races in a data-flow graph, as Figure 5 shows. Red arrows indicate potential data races. Ideally, all arrows should point forward (left to right), indicating that data is flowing forward in time. Some do, by luck. Under different conditions, they might not. When an arrow points backward, it means one thread needs data that isn't ready in another thread, so the first thread will stall.

Programmers can resolve data races by synchronizing the threads or by forcing dependent functions to execute serially. Sometimes it's a simple resource-allocation problem. In this example, the programmer tried to conserve resources by using the same memory to temporarily store all the macroblocks that the program is compressing. On a single-core processor, this approach works fine. On a multicore processor, multiple threads contend for the same memory. Figure 6 shows the data-flow graph after fixing the problems.

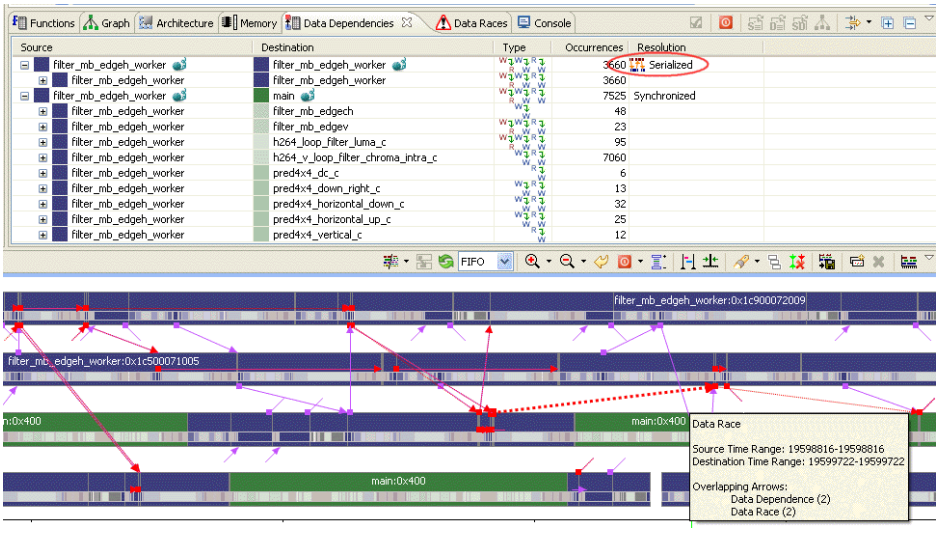

Figure 6. After the programmer forced some data-dependent operations in the edge-detection function to execute serially, Prism's data-flow graph shows all data flowing forward in time (red arrows pointing from left to right). This modification allows that function to run in parallel with other threads calling the same function.

Testing What-If Scenarios

Sometimes, adding threads or cores won't accelerate a program. There may be too many true data dependencies to take advantage of multithreading. Or perhaps the overhead of thread management outweighs the gains of executing small tasks in parallel. Or maybe the program reaches a point of diminishing returns at a particular number of processor cores. If the only way to answer these questions is to repeatedly modify the program and test it on different hardware configurations, the developers will be laboring a long time. And what if the hardware design isn't finished yet?

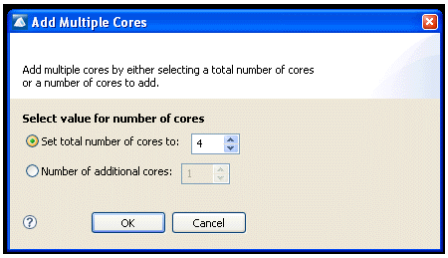

Figure 7. Software developers can quickly add or subtract processor cores to test a program with different multicore configurations. (If only hardware developers had it this easy.)

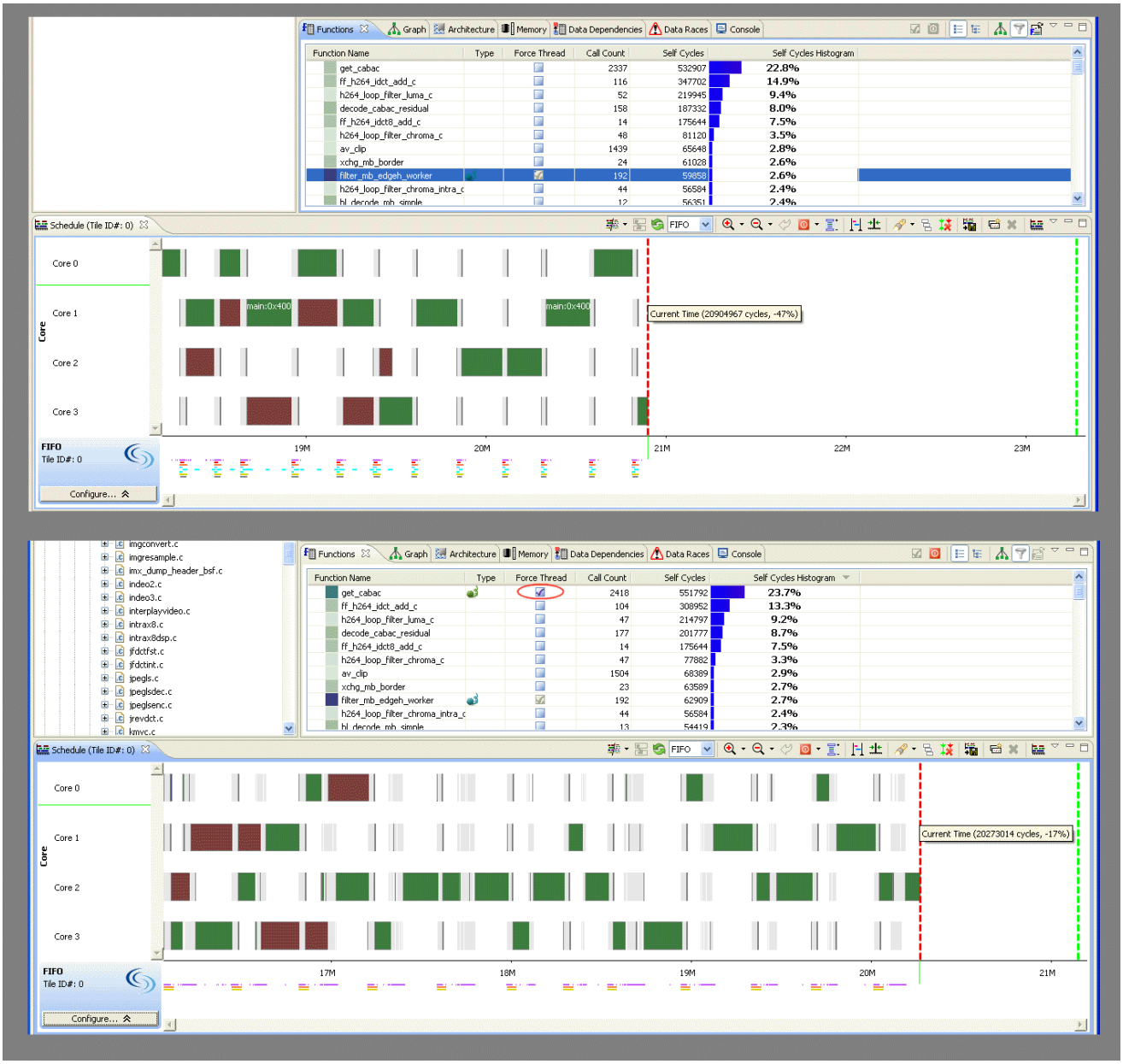

Figure 8. This composite of two cropped screen photos shows the results of forcing a function named **get_cabac** to run in parallel threads on a quad-core processor. In the top analysis, the partially threaded version of **get_cabac** executes in 47% fewer clock cycles than a sequentially coded version. In the bottom analysis, forcing **get_cabac** to run as a fully independent thread actually worsens performance—it executes in only 17% fewer clock cycles than sequential code. In this case, dependencies limit the benefits of parallelization.

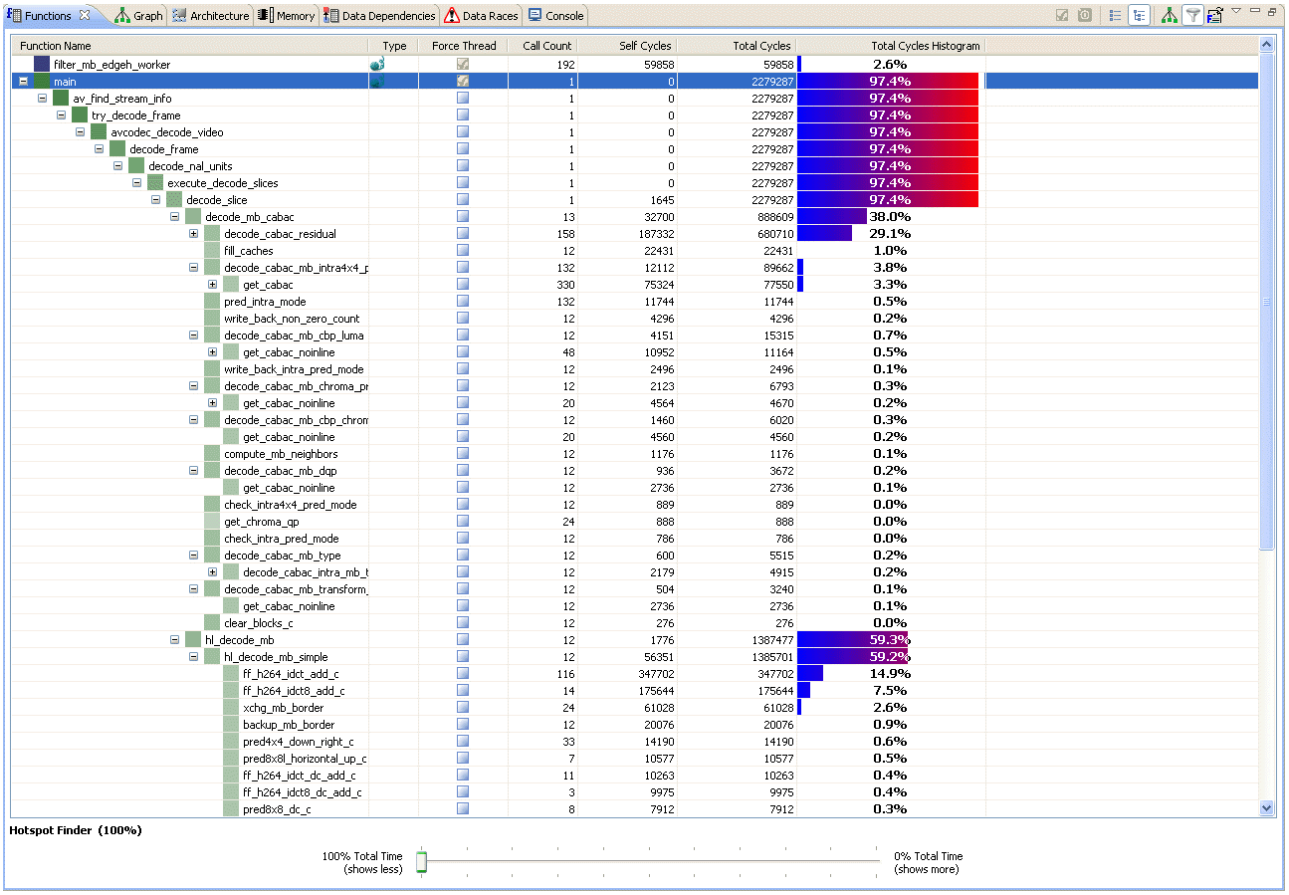

Figure 9. Prism's hot-spot finder. This screen displays all the functions of a program in a tree hierarchy and shows the number and percentage of clock cycles that each function requires during the simulation run.

Prism can save months of work by making these explorations as easy as testing financial what-if scenarios in a spreadsheet. As Figure 7 shows, adding more (virtual) processor cores requires only a few mouse clicks. CriticalBlue says that even a change this significant won't require developers to rerun the simulation. Prism reanalyzes the previously captured trace data to match the new scenario.

Similarly, developers can predict the results of executing certain tasks in separate threads—before rewriting any code. Using the original trace data, Prism recalculates all the thread scheduling and dependencies, then estimates the effect on performance. Figure 8 shows the results of forcing a function named **get_cabac** to run in its own thread on a quad-core processor. In this example, partially threaded code runs faster than unthreaded code *and* fully threaded code, because executing this function in parallel creates new dependencies that can't be resolved.

The ability to test and evaluate different scenarios without rewriting a program is huge. Developers can explore numerous possibilities and make intelligent decisions before committing human resources to refactor the code. Even if Prism leads a developer to conclude that an existing program won't benefit from any parallelization at all, the time saved is probably worth the cost of the tool.

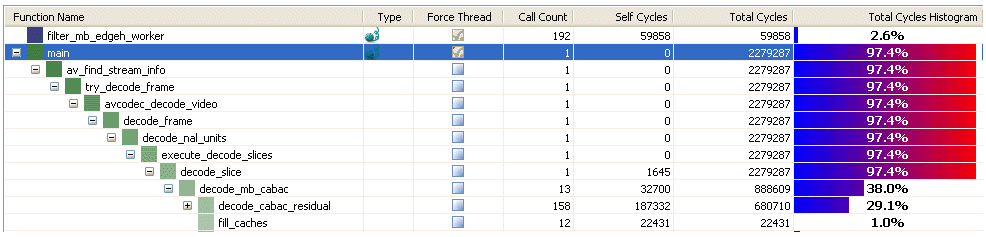

Figure 10. Prism's hot-spot finder (detail). This closeup from Figure 9 reveals that only seven functions, though called only once, account for 97.4% of the clock cycles required to encode a video stream in H.264 format. These functions are obvious candidates for optimization.

Indeed, Prism has applications beyond parallel programming. Its thorough analysis is useful for understanding the mechanics of any program and for profiling a program's performance. Prism would be valuable for analyzing any large program written years ago by

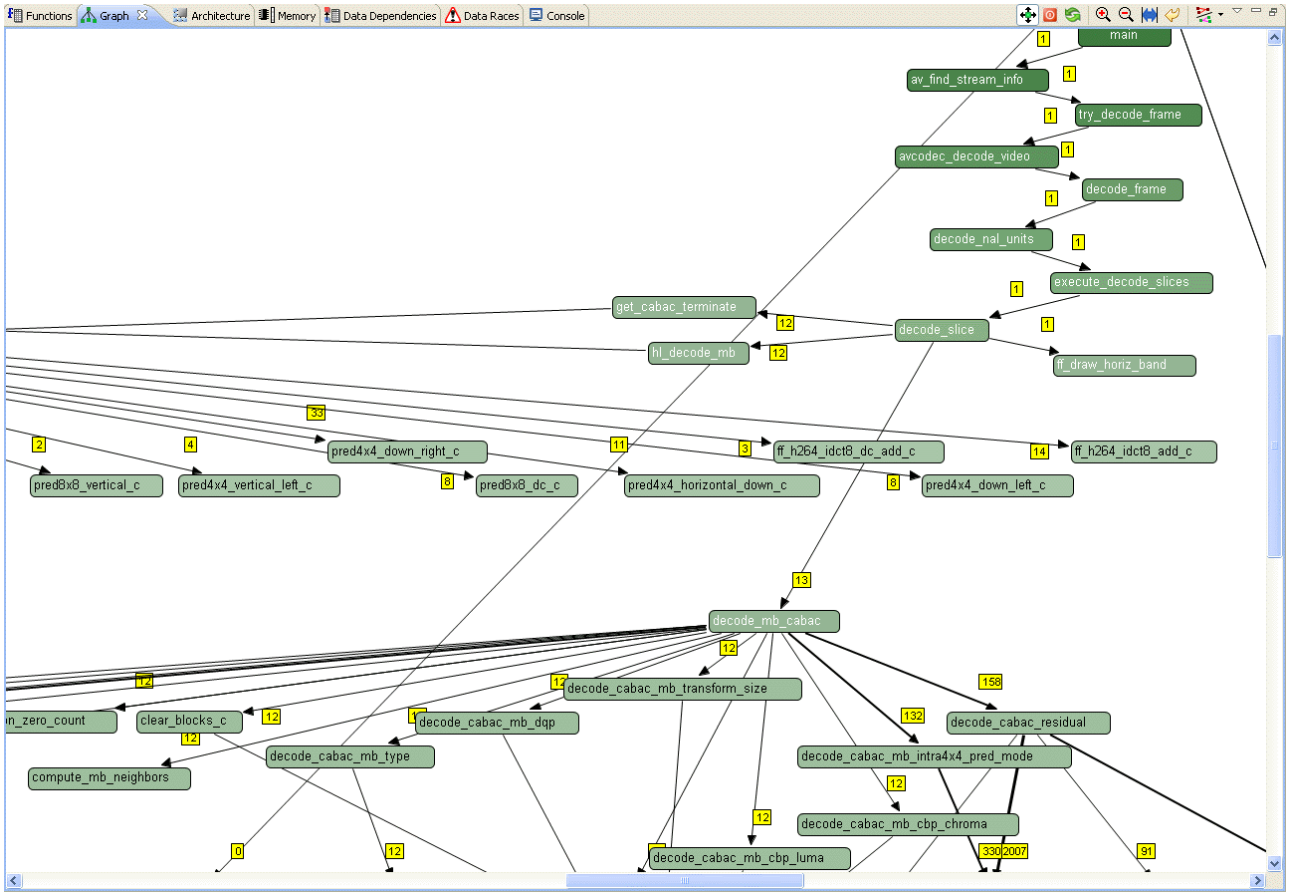

Figure 11. Prism's function-call graph. This scrollable screen displays every function in the program, with arrows connecting functions that called other functions during the simulation run. Together with the hot-spot finder shown in Figures 9 and 10, this view is another way of visualizing the hierarchical structure of a program.

programmers who have long since departed, leaving behind legacy code that must be maintained. Figures 9 and 10 show Prism's hot-spot finder, which displays a hierarchical tree of program functions and the percentage of execution time that each function consumes in simulation.

Figures 11 and 12 show another view of the same program. This time, Prism displays a graphical representation of the functions. Arrows indicate which functions are calling other functions. This view reveals long call chains that may be inefficient or create potential scheduling problems for a parallel implementation.

There is a limit to what-if scenarios. When developers finally do get around to rewriting code, any changes that significantly alter the program's behavior will naturally make the original trace less indicative of the program's new performance. Pretty soon, the original trace will become obsolete. At that point, developers must run another trace to model the new behavior and use Prism to analyze the results again. Trace runs can take a long time, especially if they are cycle-accurate instead of instruction-accurate. This iterative process adds more time to the usual development cycle of refactoring the code, recompiling the code, and testing the code.

Unfortunately, this limitation seems unavoidable. In defense, CriticalBlue says that traces can remain useful through several recompilation cycles, depending on the extent of the code changes. And with each retrace, Prism performs a fresh analysis, looking for new ways to optimize the program.

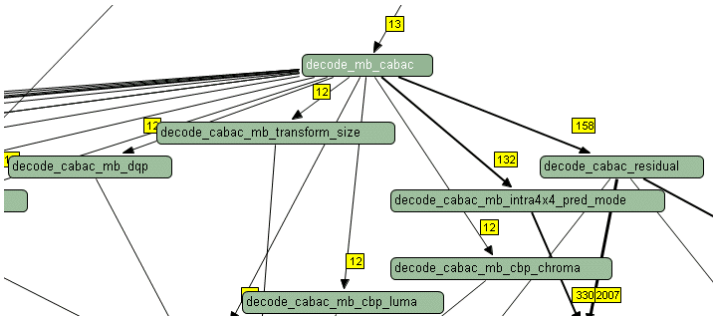

Figure 12. Prism's function-call graph (detail). This closeup from Figure 11 shows individual functions in green boxes. Arrows indicate function calls. Yellow boxes record the number of times the program called a particular function during the simulation run.

Price & Availability

CriticalBlue's Prism code-analysis tool is available now for ARM and MIPS embedded-processor cores. CriticalBlue has demonstrated beta versions for Freescale's Power Architecture cores and the Renesas SH-Mobile architecture.

Prism is available in three forms of platform support packages (PSP). A Core PSP analyzes dynamic trace data for one program running on a single-core processor. A System PSP can analyze more system-level behavior for one program running on a platform, which may include a multicore chip. CriticalBlue can tailor a Custom PSP for a customer's requirements. Prices start at $200 a month to lease a Core PSP for a target processor. Prices for a System PSP or Custom PSP are negotiable.

Prism requires a software simulator for the target processor. Simulators are available separately from processor vendors and third parties.

For more information about Prism, see: www.criticalblue.com/criticalblue_products/prism.shtml

CriticalBlue Finds Early Adopters

Although Prism was officially released at the Multicore Expo in March, CriticalBlue began delivering early versions to lead customers in June 2008. Toshiba (an investor in CriticalBlue) was the first customer. Toshiba is shipping a custom version of Prism (V-Prism) as part of the software development kit (SDK) for its Venezia platform. Venezia is a homogeneous multicore SoC architecture for embedded systems. It's built around Toshiba's Media Embedded Processor (MeP) cores. (See *MPR 6/10/02-02*, "New Processors for New Media.")

For other customers, CriticalBlue delivers Prism in a platform support package (PSP) for specific processors. One form is a Core PSP, which typically uses an instruction-accurate simulator to analyze a single program running on a single core. It's intended to help programmers implement multithreading across multiple cores.

Another Prism package is a System PSP. It is specific to a processor-based platform—such as a multicore chip, a multiprocessor system board, or a processor family. CriticalBlue says a System PSP models the system architecture more closely, includes more system effects in the trace, and provides more analysis and suggestions specific to the target platform.

Currently, Prism can analyze the trace of only one program at a time. Other programs can run during the trace, and they will affect Prism's analysis as they compete for resources and interact with the traced program. But Prism's detailed trace analysis will be limited to the target program. CriticalBlue says a full trace analysis of multiple programs running simultaneously is feasible, but it's not currently in development.

A third option for Prism is a Custom PSP. On demand, CriticalBlue can support almost any processor or platform, as the company has done for Toshiba's Venezia. In addition, developers can use Prism alongside Cascade, CriticalBlue's aforementioned C-to-RTL tool (available separately). With both tools, developers can both optimize their software and generate hardware coprocessors to accelerate critical functions.

At present, CriticalBlue has PSPs for the ARM and MIPS architectures, including the ARM Cortex-A9 MPCore, ARM11 MPCore, MIPS 1004K, and MIPS 74K processors. CriticalBlue has demonstrated beta versions of Prism for Freescale Power Architecture processors and the Renesas SH-Mobile architecture. PSPs for other embedded processors are in development. Pricing starts at $200 a month for a Core PSP.

Just because Prism doesn't currently support a particular processor core or CPU architecture doesn't mean it is useless to other developers. CriticalBlue says some developers are using the ARM version of Prism as a general-purpose tool for analyzing their program code, even though they aren't targeting ARM. The analysis is, necessarily, rather coarse. But it can lead programmers to a better understanding of their code and help them find opportunities for parallelism.

Prism's Potential Limitations

An optical-glass prism is a wondrous object that reveals the hidden color components of white light. Similarly, CriticalBlue's Prism reveals the hidden aspects of a program. Interactions that are unimportant when code executes sequentially become very important when the same code tries to execute in parallel. Without a tool like Prism, data dependencies may remain as invisible as the spectrum of colors concealed in white light. So, in many ways, Prism can be educational as well as analytical. It's a valuable learning tool for the vast majority of programmers who weren't formally schooled in parallel programming and who need to update their skills.

In practice, some limitations are bound to surface. One is that any analysis based on dynamic trace data depends on the nature of the trace. The fewer possible paths through a program, the better Prism's analysis is likely to be, because there is less data to analyze and fewer execution scenarios. When a program has many possible paths of execution, not only will a fully explored simulation take more time, but it may also exhibit different patterns of dependencies.

As an analogy, consider a GPS trace of an automobile journey. There may be numerous possible routes from start to finish, each with its own hazards and potential bottlenecks. To find the best route, a driver may have to explore several routes, then do a comparative analysis of the GPS

data. Although Prism can reanalyze a trace for different scenarios, additional simulation runs may still be necessary in some circumstances.

A systemwide analysis of multiple programs running simultaneously is even more challenging, especially if the programs contend for shared resources. Even the operating system becomes a factor. In a shared-memory SMP system, the OS and application programs compete for the same resources. The number of possible interdependencies rises with each additional program, thread, and processor core. Even with a tool as powerful as Prism, developers can be overwhelmed—especially because Prism currently analyzes only one program trace at a time.

Another limitation of Prism is that it can't be more cycle-accurate than the simulator capturing the trace. Some dependencies are sensitive to scheduling conflicts that only a fully cycle-accurate simulation can expose. Cycle-accurate simulators are agonizingly slow, particularly when simulating SMP on a multicore processor. On the other hand, Prism can perform a great deal of analysis on a cycle-accurate trace, reducing the need to rerun the trace after each recompilation.

We have already mentioned that Prism is currently limited to only a few CPU architectures—though we agree that ARM, MIPS, and the Power Architecture are the best places to start. And for now, Prism can't perform a system-scale analysis of a heterogeneous multicore design, which rules out many embedded SoCs. But CriticalBlue seems eager for a challenge. Prism is very much a work in progress, and CriticalBlue says it will entertain almost any proposal from an important customer.

For More Information

For related information about multicore processors and software-development tools, see the following *Microprocessor Report* articles:

MPR 12/22/08-01, "AMD's Stream Becomes a River"
MPR 7/28/08-01, "EEMBC's MultiBench Arrives"
MPR 7/28/08-02, "Editorial: Tools for Multicore Processors"
MPR 4/28/08-01, "Multicore Multithreading With MIPS"
MPR 3/31/08-01, "Editorial: Think Parallel"
MPR 1/28/08-01, "Parallel Processing With CUDA"
MPR 12/31/07-02, "Editorial: The Future of Multicore Processors"
MPR 11/26/07-01, "Parallel Processing For the x86"
MPR 8/13/07-01, "Fujitsu Calls Asynchronously"
MPR 6/4/07-01, "MIPS 74K Performance Update"
MPR 5/29/07-01, "MIPS 74K Goes Superscalar"
MPR 4/30/07-02, "Editorial: The Dread of Threads"
MPR 10/2/06-01, "Number Crunching With GPUs"
MPR 5/24/04-01, "ARM Opens Up to SMP"

After seeing what Prism can do, despite the drawbacks, it's hard to imagine tackling any but the simplest parallel-programming project without such a tool. Nobody said software development in the multicore era would be painless.