criticalblue

Prism

Rapid Parallelization of Embedded Software

Part One: Understanding Sequential Code

The new CoreMark benchmark from EEMBC is designed to evaluate the performance of embedded processors by modelling real world workloads. CoreMark includes 3 major algorithms looking at core characteristics:

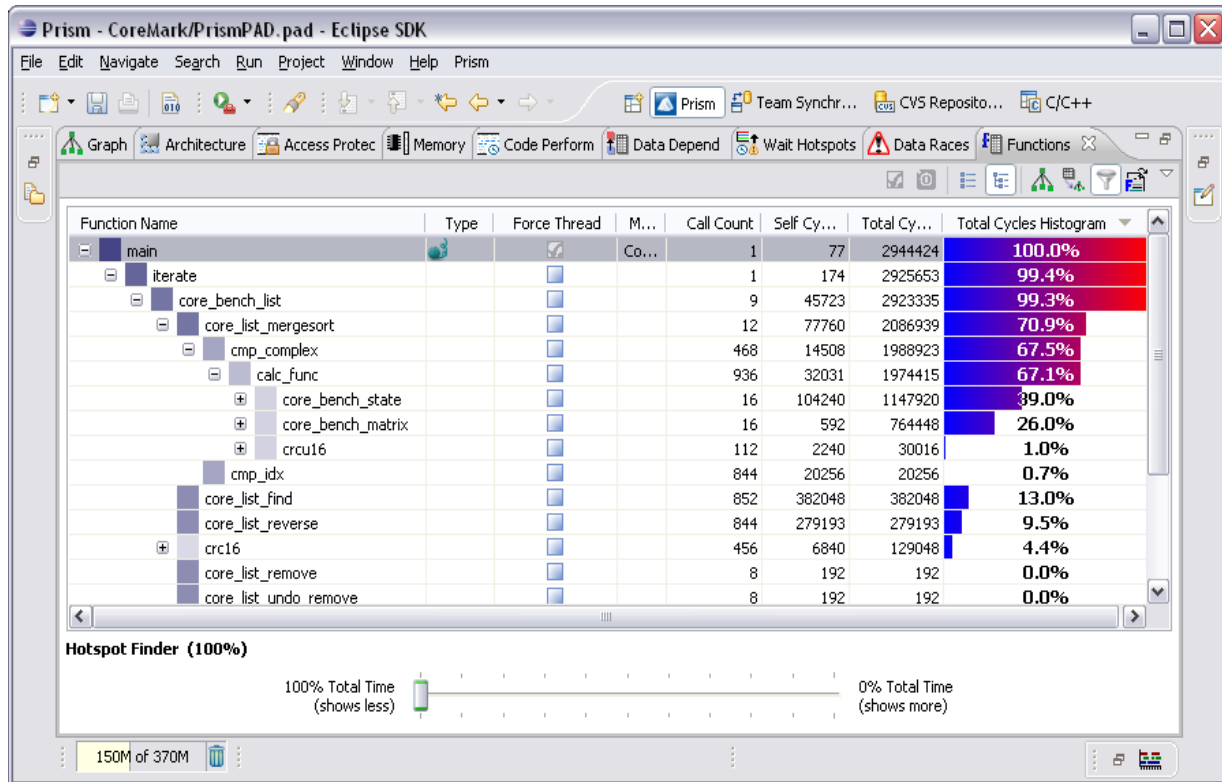
- List manipulation - pointers and data access through pointers.
- Matrix manipulation - serial data access and potentially using ILP.
- Simple state machine - FSM that exercises the branch unit in the pipeline.

Although it is strictly against the rules to make any changes to the source code for the benchmark, such a collection of highly representative algorithms provides a good, compact example on which to demonstrate how Prism can be applied to real applications.

Prism is the software analysis tool that allows the user to quickly understand the structure, behaviour and runtime profile of sequential software and assists in the exploration of parallelization strategies when migrating to multicore platforms.

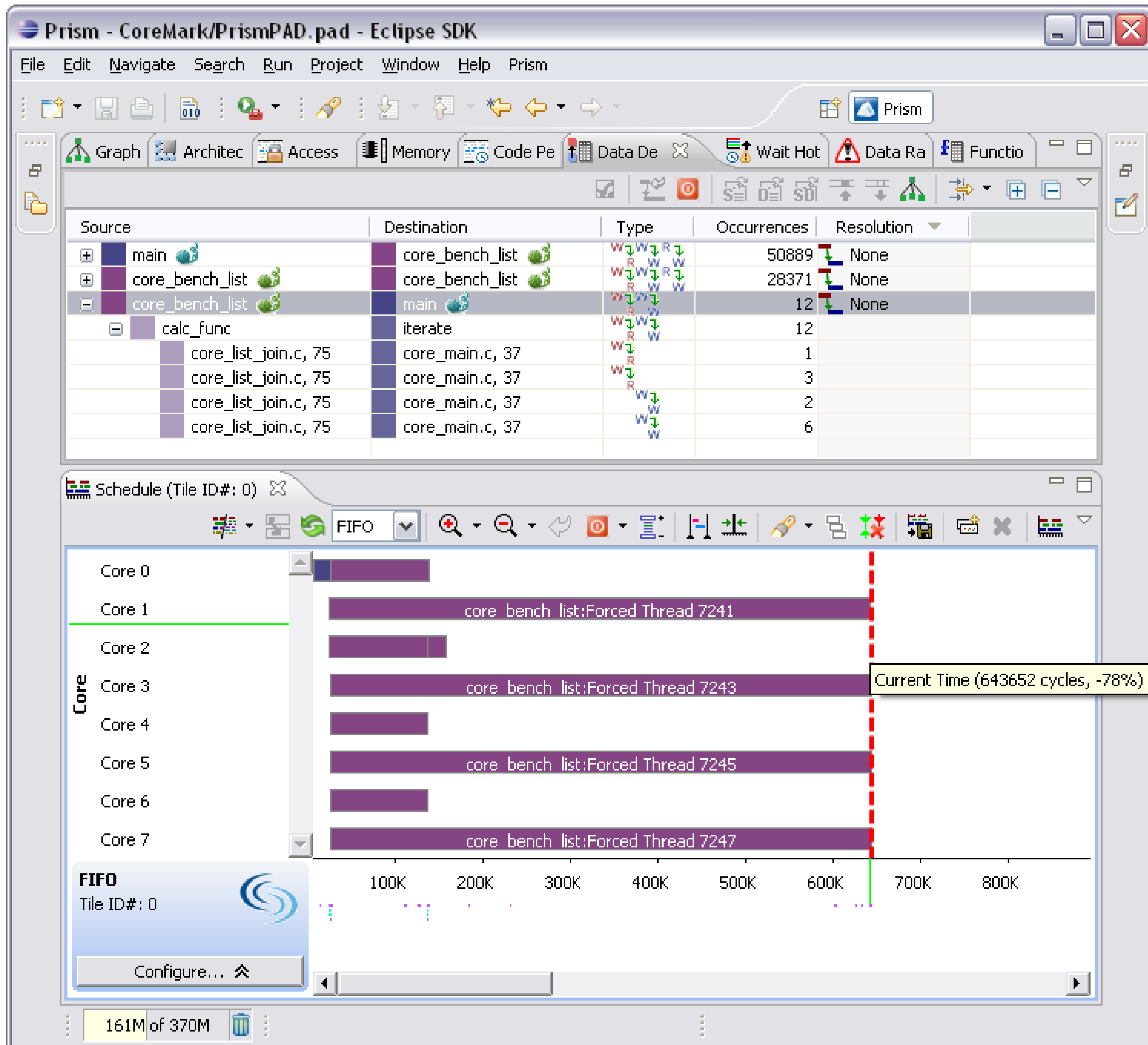
Prism analysis starts by capturing a trace of the benchmark as it executes. Once the trace is loaded, the profile of the sequential application can be examined in detail as shown in the screen shot below. The view shows the function profile and, using Prism's ability to map rows in the call tree profile to lines in the source code, it quickly becomes apparent how the benchmark works.

The three algorithms are implemented by the functions `core_bench_list`, `core_bench_matrix` and `core_bench_state` and are integrated together in an interesting way. Instead of simply doing running several iterations of each algorithm in turn, CoreMark iterates over a list in `core_bench_list` which then uses the other 2 algorithms to calculate the comparison values for the merge sort part of the list benchmark (in `cmp_complex` called from `core_list_mergesort`). The matrix and state machine benchmark routines are called an equal number of times although `core_bench_state` has a significantly longer runtime than `core_bench_matrix`.



Using this information, the user can start to make decisions about how the application might be broken down into threads to take advantage of a parallel architecture.

Prism allows you evaluate the impact of adding threads *without* modifying the code via its **Force Thread** feature which simple tells Prism to treat calls to specified functions as if they where run in separate threads. Applying this approach to the `core_bench_list` function in the benchmark results in the following schedule across 8 cores; as modelled and visualized within Prism.

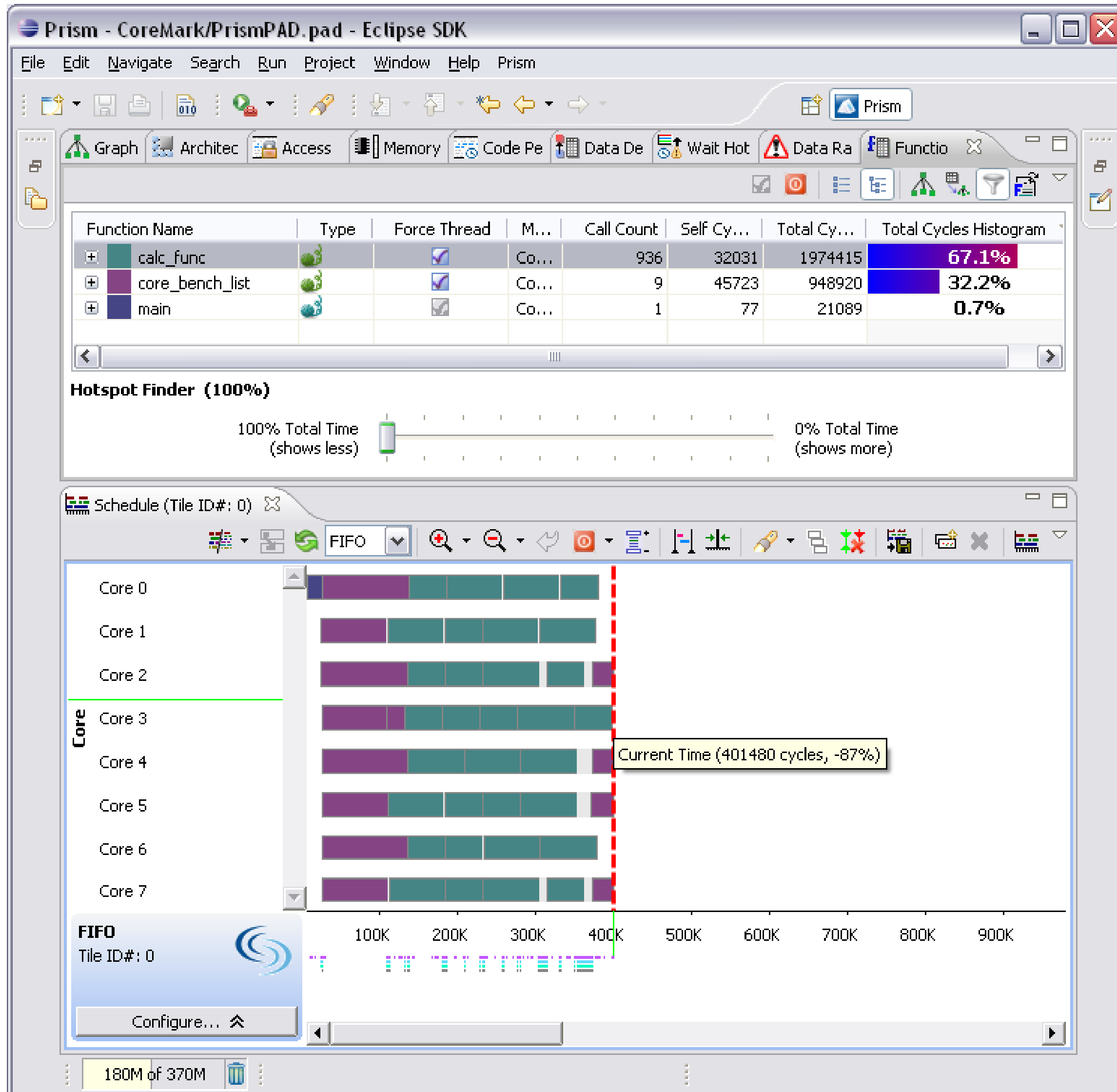


The above view tells us a few things about this threading strategy. Firstly, in the top half of the screen, Prism identifies all of the dependencies which exist between the threads we are introducing and any existing threads in the system (here `main` is the only other thread). If we want run `core_bench_list` efficiently and safely without data races the code must be changed to remove or manage these dependencies.

Since this is a major undertaking, Prism allows you to model the impact of such changes in advance so the user can be sure the effort is worth it. In the example above, the dependencies have been disabled (by setting the Resolution column to 'None') which has exposed substantial parallelism in the schedule shown in the bottom half of the screen.

Second, the schedule shows that we are going to be limited in the amount of speedup we get due to the runtime of each thread. This schedule indicates that we would expect more cores to be able to process more iterations of the benchmark in the same amount of time but the time *per iteration* will not scale with the number of cores. So it won't scale up if there is a constant amount of work to do per unit of time.

Further thread forcing of `calc_func`, which is lower in the call tree, leads to the following schedule which appears to scale much more effectively as can be seen below. Again the data dependency view tells us what needs to be changed in order to achieve these results.



At this point we have a good idea what is going to happen on our 8 core example but how effective is this approach across a range of architectures? Will we get the same kind of performance on 6? Using Prism's Application Analysis feature we can generate a table of results for different numbers of cores as shown below to answer these kinds of questions.

Prism - CoreMark/PrismPAD.pad - Eclipse SDK

File Edit Navigate Search Project Run Window Help Prism

PrismPAD.pad

Application Analysis

Application analysis calculates some summary information after applying the current core, thread forcing, and dependency resolution settings for the schedule selected below.
Note that this operation may take a long time.

Select Scheduler FIFO

Core count to analyze

Use current architecture | Force all architecture cores to be enabled for this analysis

Select core count 12 | Sweep across core range from 1 to specfied core count

Run Analysis | Export Analysis

Last analysis run took 6 min 9 sec

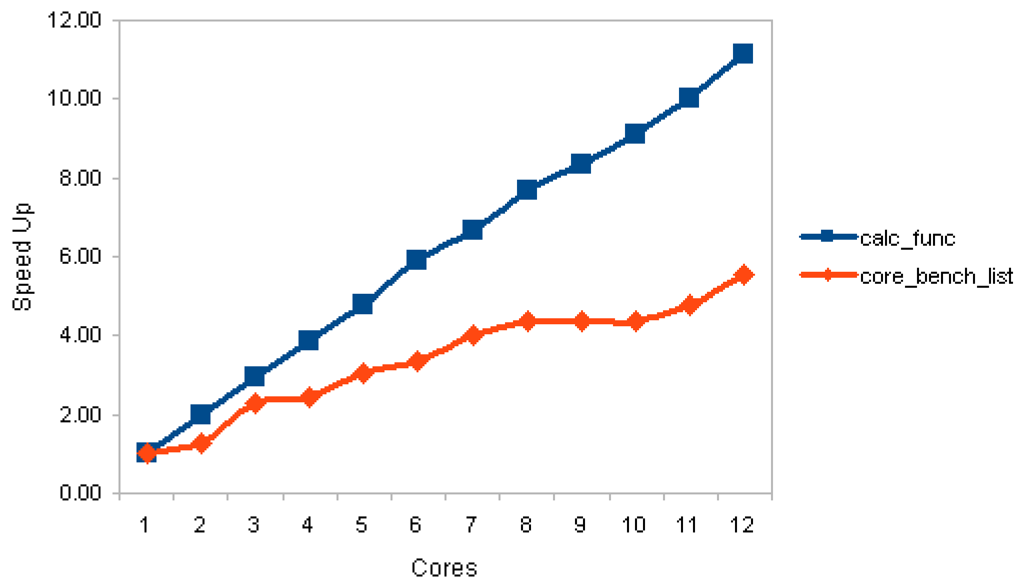
Core Count	Schedule Time	Percentage Change	Core 0* Utilizat...	Core 1* Utilizat...	Core 2* Utilizat...	Core 3* Util
1	21848633	0.0%	99%			
2	10975286	-50.0%	99%	99%		
3	7399373	-66.0%	99%	97%	97%	
4	5600231	-74.0%	98%	97%	97%	96%
5	4549878	-79.0%	98%	95%	96%	94%
6	3804984	-83.0%	96%	97%	95%	95%
7	3283874	-85.0%	97%	93%	95%	95%
8	2880340	-87.0%	96%	92%	95%	96%
9	2596397	-88.0%	97%	94%	94%	91%
10	2375815	-89.0%	95%	94%	92%	91%
11	2196195	-90.0%	96%	93%	92%	88%
12	2046631	-91.0%	95%	93%	91%	87%

Prism Properties | Histogram | App. Analysis

140M of 334M

The results of the sweep report not only the overall performance estimate but also the utilization of each of the cores which is a good indication of the overall efficiency of the schedule for a given number of cores particularly when considering the total power consumption of the system.

Once generated, the information can be easily exported into a spreadsheet package for more in depth analysis. The graphs below show the results for the two threading approaches we have discussed across a range of cores with the reductions in cycles converted into speed up compared to the sequential performance.



Now, after only a few hours of analysis, we know how the code is structured, which functions we want to run in threads and how much refactoring of the code is required in order to ensure the parallelism is introduced in an efficient and safe manner.

Once the implementation is under way, Prism can be used further to verify the threaded code by detecting accidentally introduced data races and identifying inefficient implementation decisions which may lead to resource or lock contention slowing down the execution of the program.

We will take a closer look at these features, and how Prism guides the user to the code requiring refactoring to resolve data dependencies , in the next instalment.