You Can't Parallelize Just Once: Managing Manycore Diversity

Position paper for the Workshop on Manycore Computing at ICS'07

David A. Penry
Department of Electrical and Computer Engineering
Brigham Young University

One of the greatest challenges for the use of manycore architectures will be the growing diversity of manycore systems. This diversity will come in many forms:

- **Architecture**. Products will have a wide range in the number of cores, mix of cores, specialized units, and communication characteristics. This variability will occur not just between products, but between individual chips as well; increased process variability will require yield enhancement techniques which shut down cores which are defective or rate different cores for different frequencies.
- **Goals**. The abundance of cores will be used for more purposes than just increasing the throughput of embarassingly parallel applications. These goals may include reduced latency for a specific application, fault tolerance, or availability.
- **Programming languages**. There will not be "one language to rule them all". Applications will be written in a variety of languages with very different run-time characteristics. Some languages (e.g. data parallel languages) will provide explicit information about the parallel semantics of the program. Others, such as C and C++, *which will not go away*, will not provide such clear information.
- **Pre-parallelization.** While many important applications will be written in a parallel fashion, the world cannot afford the enormous human effort required to hand-paallelize all software. Furthermore, legacy binaries will continue to be used. But users will expect *all* of their applications to benefit from manycore architectures.
- **Dynamicism.** Systems will not retain fixed characteristics over time. Manycore chips will have mechanisms for disabling cores when they fail. Perioding re-evaluation of core timing will occur. Power regulation mechanisms will change processor speeds or disable cores frequently. In addition, variations in system load when a chip is not dedicated to a single application (a trend exacerbated by virtualization) make the desired degree of parallelization for a given mechanism vary over time.

How can these vast differences be managed? More specifically, how can a software developer ship an optimized, parallelized software product when the target system is so unpredictable? No one parallelization will perform well for all systems. How many versions of the application must a vendor ship? Tens? Hundreds? Thousands?

We argue that the most managable approach to such diversity is to delay optimization and parallelization until runtime. Such an approach has the following characteristics:

1. Th software vendor ships a single product "binary". This "binary" is not final machine code, but an IR representation which preserves types and information about parallel semantics which the original program has (optionally) supplied. Legacy binaries can be supported through a binary translation process which outputs this IR format (with reduced type information).

2. The runtime system operates as a parallelizing, optimizing, just-in-time compiler. This JIT includes both traditional parallelization techniques as well as newer pipeline-based and speculative parallelization techniques. It knows the hardware characteristics of the system upon

which it runs and can tune its parallelizations to those characteristics. Indications of parallel semantics in the IR are exploited. Note that the JIT uses this parallel semantic information but is not bound by it; it may *reparallelize* the program to better target it to the hardware. The JIT may also improve the program along dimensions other than performance through techniques such as software-based fault-tolerance or redundand multithreading. Note also that a JIT obtains all the standard benefits of runtime optimization, e.g.:
    - Many programs call dynamically linked libraries. Static compilation is unable to optimize or specialize the library code for a given application, while runtime parallelization and optimization can.
    - Dynamic code specialization can improve code which uses "run-time constants".

3. The JIT does not parallelize the application just once at load time; it periodically tunes the parallelization for current system load conditions, providing *dynamic reparallelization*. Some benefits of dynamic reparallelization include:
    - Task-scheduling based schemes require accurate task costs to produce good schedules; run-time measurements are much better than static estimates.
    - Speculative techniques often have high costs for mis-speculation. By tracking the mis-speculations actually incurred, the JIT can remove poorly performing speculation.

Applications cannot be parallelized just once; they require separate parallelization targeted to the actual hardware and environment upon which they will execute. The diversity of the hardware, the applications, and the execution environments will grow enormously for manycore chips, making it impossible for software vendors to manage the different parallelized versions of an application. Deferring the parallelization and optimization to a runtime system will allow vendors to concentrate upon the functionality and features of the application without worrying about exactly how the parallelization will occur for a plethora of systems. The runtime system will handle the details. As a result, users will experience the increased application performance and/or reliability they will expect from manycore systems.