Data Flow Compilation for Greater Parallelism

Jeff Mason , Dave Bennett, Tom Ace, Arvind Sudarsanam
CPU Technology
1500 Kansas Ave, Suite #3d
Longmont Co, 80501
(jmason,benzene,tace,asudarsanam)@cputech.com

Introduction

Modern computation is mostly performed on one or more Chip Multi Processors (CMPs). CMPs contain multiple cores (or processors) that each execute a separate stream of machine instructions. Modern compilation techniques that transform programmer source code into one or more streams of machine instructions mostly follow concepts developed long ago to transform the exact commands the user provided, while maintaining their sequence of execution. Dataflow Computation was a computer architecture methodology developed in the 1970s and 1980s [1]. In contrast to traditional sequential compilation, Dataflow Compilation attempted to create independent compute blocks that would execute whenever all of their inputs are available. This methodology offered the potential to extract far more parallelism in a given code than was possible to be created by a programmer. It was eventually abandoned due to the difficulty of implementation on the computer architectures of the time. Thirty years of continued development in computer architectures has brought us CMPs with large amounts of faster memory, memory caches, pipelining and faster overall computation.

The paper below lays out three changes that could be made to modern compilation technologies and computer architectures to allow the use of dataflow methodologies to better utilize CMPs. Both software and CMP architecture changes are suggested. Following are the proposed changes that are discussed in the remainder of the paper:

1. Develop a new Dataflow Compilation toolset that transforms a given sequential code into multiple compute blocks based on dataflow analysis. This allows for greater parallelism across cores in a CMP.
2. Disable the use of each core-specific level1 (L1) cache for all non-stack/non-constant data access. This allows for removal of the non-trivial overhead of cache coherency. Replace the single level2 (L2) cache that is shared between cores with a Crossbar Connected Cache (CCC) [2]. This will enable more concurrent memory accesses to occur between cores.
3. Create a mechanism for direct inter-core communication of small amounts of data that does not affect normal shared data access. This would allow for more efficient communication of temporary partial results.

Dataflow Compilation

Dataflow architectures were developed and examined over 30 years ago [1]. The main idea was that a given sequential program as written by any programmer could be reformed into a set of computation blocks whose only criteria for execution was the availability of its inputs. This is how a spreadsheet with formulas behaves. When a given value is changed, all of the other formulas/values that rely upon that value are updated. This continues with new updated values until no more updates can occur. Dataflow Compilation attempted to convert programmer source code into such a form. Some problems that plagued Dataflow Compilation were the requirement that memory be kept consistent (e.g. a write of a value needed to occur before that value could be read) and the difficulty of keeping multiple values provided to a computation block such that the correct outputs went to the correct consumer block. At the time these problems could be solved only at the expense of large amounts of memory and extremely great communication penalties which made it too slow to be a viable option.

CMPs offer larger memory spaces and much faster communication between cores. The ability to more concisely transform a sequential program into a number of computation blocks is also more accessible due to the complexity and vastness of compilers. Using a revamped Dataflow Compilation methodology it is possible to transform any given sequential program into a set of more efficient computation blocks that fully represent the structure of the input code. Memory consistency is maintained by the addition of synchronization information that can be passed between these computation blocks. The possibility that each of these compute blocks can execute on different cores allows greater parallelism and speedups than currently available with traditional techniques. Dataflow Compilation also allows for the generated instruction streams to be manipulated in ways not available to traditional sequential compilation. The communication costs of data is still the largest deterrent to this methodology. Solutions to the communication cost problem would greatly enhance the ability of Dataflow Compilation to reduce runtimes of most if not all programs running on CMPs.

Memory cache restructuring at the hardware level

Figure 1 represents a generalized view of an ARM Cortex A-9 core. More silicon area of this core is dedicated to cache memory than to computation.

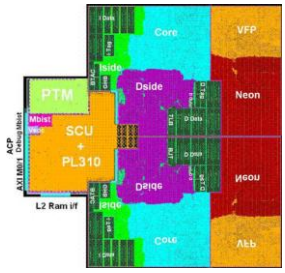

Figure 1
ARM Cortex A-9

Modern cores are based upon a design developed and well suited towards single core computation. Each core contains its own L1 cache for faster execution. Cache memories are local SRAM connected directly to the compute fabric that hold recently accessed data or instruction values used to speed their later access. Caches have been shown to dramatically accelerate overall processing due to the slow offchip DRAM access times. However incorporating this caching technology into a CMP creates the greater problem of maintaining coherency between the individual core caches when common values are modified. The cost of maintaining cache coherency is at the hardware level (additional detection circuitry) and at the execution level (invalidation of cache lines). Since the action of updating caches must happen between cores during memory access, it is built into the hardware and not visible to nor under software control. This makes it extremely difficult to determine how much these actions are affecting the operation of the chip. In a CMP where the data in a given core is highly connected to the data in another core (integral to Dataflow Compilation above) it is conceivable that the run time savings of using the L1 cache is outweighed by coherency overhead. Disabling the L1 cache for shared data access would alleviate this problem. Leaving the L1 caching mechanism in place for other accesses (e.g. instructions, constant data) would continue to have benefits.

Disabling core specific L1 cache for shared (or potentially shared) data access puts a greater burden on the L2 cache. Current L2 cache architectures closely mirror L1 cache architectures in that they are designed for single core access. When multiple cores have the potential to simultaneously access the L2 cache this type of architectures can greatly slow overall operation. If the single shared L2 cache on a CMP was designed more along the lines of Crossbar Connected Cache [2] then multiple accesses from independent cores could be more easily supported. These two changes to the architecture of a CMP could greatly enhance its ability to process parallel executing code. The amount of sharing enabled by Dataflow Compilation would greatly benefit from these changes.

Direct core to core communication

In current CMP systems a single byte of communication from one core to another is always treated as a shared memory access. This happens even if the data is a temporary intermediate value written by one core and read by a different core. This type of communication uses the standard caching mechanisms defined above. Even allowing for a reworking of the L1/L2 caches as defined above, the overhead for such communication far outweighs the value of such small amounts of communication. The addition of circuitry to allow for direct core to core communication that bypasses the standard shared data caching mechanism would greatly lower the latency for such low bandwidth communication. Lowering this latency could make it much more useful in general computing. The Dataflow Compilation model explained above would greatly benefit from such changes. This would also free up bandwidth in the L2 cache for true shared data.

Conclusion

This paper proposed three changes that can be made to the current CMP hardware architectures and software methodologies to more efficiently take advantage of multi-core processors. These changes, though non-trivial from the hardware and tool developer point of view, would not require the development, training and use of a new software programming paradigm to more efficiently use CMPS as often proposed. Changing compute hardware and compilation methodologies is far easier than changing users.

References

[1] Dennis, Jack B. "A preliminary architecture for a basic data-flow processor", *Proceedings of the 2nd Annual Symposium on Computer Architecture*, 1975

[2] Lin Li, N. Vijaykrishnan, Mahmut Kandemir, Mary Jane Irwin and Ismail Kadayif, "CCC: Crossbar Connected Caches for Reducing Energy Consumption of On-Chip Multiprocessors", *Dept. of Computer Science and Engineering*; Pennsylvania State University, 2003