Computing With Billion Transistor Chips

Guri Sohi
Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI 53706

1. The Computing Problem

The goal of more powerful computers is to enable more demanding applications. I will not venture to take a guess at what will be the dominant "killer applications" for billion transistor chips. Historically new applications have arisen as computers became more powerful; I expect the future to be no different. New applications will continue to demand more and more computing power. The problem of computing is then one of determining how to provide a desired level of computing power given the available technologies.

There are three broad aspects of the computing function that I will comment on in this paper: (i) Algorithms for carrying out the desired computation, (ii) Hardware to perform the desired operations, and (iii) Software (compilers) that converts a computation expressed in a high-level language to a form that hardware can operate upon. In practice these aspects are related in complex ways, and can't be considered in isolation. I will start by briefly reviewing these aspects of the computing problem. Then I will comment on the hardware components of billion transistor chips, and then proceed to the potential impact on software and algorithms.

Applications are described in a high-level language program, important parts of which include the *algorithm* for carrying out the desired computation, and the *data structures* used to represent the information. The resulting program is a compact (static) representation of the operations that have to be carried out. An underlying assumption is that a *sequencer* will sequence through the static program to recreate the dynamic operations that have to be carried out. An important issue is the number of sequencers: a program created with the assumption of a single sequencer is a *sequential* program; a program which explicitly states that multiple sequencers could be used is a *parallel* program.

A program written in a high-level language has to be converted into another static program that is understood by the underlying hardware. I will get back to this after commenting on the functions that hardware has to perform. The hardware takes the static program (along with an input data set), *sequences* through it to determine the dynamic sequence of operations to be executed on the input data, and *executes* the operations, creating the desired output. As operations are executed, values are created for use by later operations. This *inter-operation communication* is carried out using a convention agreed upon when the program was created: producer operations bind a value to a (named) storage location; consumer operations access the value by reading from the storage location. More powerful computers means more powerful sequencing, operation execution, and inter-operation communication; carrying out these functions in a parallel fashion is an obvious way of increasing their power.

Programs are written in high-level languages, the most popular ones of which are sequential. In the process of converting a (sequential) high-level language program to a form understandable by the hardware, compilers try to analyze the program and reason about its behavior, possibly transforming it so that better use can be made of the execution hardware. In particular, parallelizing compilers attempt to analyze the program to see if it can be "automatically parallelized" i.e., converted from a sequential form to one that has multiple sequencers inherent in it. This automatic parallelizing process is perhaps the most involved aspect of the compilation process today.

Before I present my view of how the above will be carried out in the billion transistor chip era, it is instructive to revisit the RISC-era of the 1980s, i.e., the 100K-1M transistor chip era. At that point, hardware resources were at a premium, the benchmark machine of the era (the VAX-11/780) was sequencing through the program at a (sustained) rate of 1 instruction every 10 cycles. An effort was made to improve the sequencing rate to 1 instruction every cycle; pipelining was the method of choice. Instruction sets were crafted to facilitate pipelining. Moreover, since on-chip resources were limited, instruction sets were designed to be implementable with minimal execution and control hardware. Processing hardware was simplified, and compiler software was given the task of optimizing performance.

2. Computing Hardware

The microarchitecture of a billion transistor processing chip will carry out the basic hardware functions: sequencing, operation execution, and inter-operation communication, in a more powerful manner so that higher performance can be achieved; after pipelining, more parallel execution is the obvious method.

An important issue in the choice of the microarchitecture will be its "interconnect requirement", i.e., the number (and size) of wires required to connect the various components -- wires are likely to be much more important than logic functions, both for speed as well as (die size) cost, in billion transistor chips. In addition, many practical considerations will influence the microarchitecture choice, three of the important ones are: design (and validation) complexity, complexity of the testing process, and power consumption. The considerations are going to be of paramount importance, perhaps even dominating the "performance" attributes in the choice of a microarchitecture.

The above considerations will necessitate microarchitectures that are constructed by replicating simple, regular, microarchitectural components. The complexity of the design and validation process is reduced if the number of distinct components is reduced, and if the components of different generations of microarchitectures are similar. Likewise, the complexity of the testing process is reduced if hierarchical techniques can span all levels of the microarchitecture. In the power arena, I believe that the ability to have varying power-performance characteristics will be important. That is, the microarchitecture should be such that portions of it can be turned off, if power savings are desired, at the expense of performance. For higher performance, all the resources of the microarchitecture could be applied, consuming more power in the process. Microarchitectures built up from replicated components some of which could be turned off and still allow correct, albeit low performance operation, facilitate variable power-performance. Finally, reducing wire lengths and the number of wires also dictates the use of a microarchitecture with multiple replicated resources. Here, inter-operation communication that is "local" to a set of resources could be carried out with local wires, with non-local wires being used for (hopefully less frequent) non-local communication.

I now comment on how I expect the hardware to be carrying out the functions of sequencing, operation execution, and memory inter-operation communication.

2.1. Sequencing

There are two options to improving the power of the sequencing process: (i) wider single sequencers, and (ii) multiple sequencers. Current wisdom has it that multiple sequencer microarchitectures can (theoretically) extract more parallelism than single, wider sequencers [2]. With better branch predictors, and with the use of predicated execution, no doubt the parallelism-extracting abilities of single-sequencer microarchitectures will continue to improve, possibly even matching the parallelism-extracting potential of multiple sequencer microarchitectures. However, the issues mentioned above will dictate multiple sequencer microarchitectures (or at least microarchitectures that have more in common with multiple sequencer microarchitectures than with single sequencer microarchitectures).

The multiple sequencers could either be applied to sequence through a parallel program, as in a multiprocessor microarchitecture, or through a sequential program, as in a Multiscalar microarchitecture [6]. The latter case requires considerable support for speculative execution of all forms; more on this in the next section.

An enhancement to the sequencing process is what I call an *informed sequencer*. With a traditional sequencer, the sequencing/decoding process knows about the instruction it is sequencing through currently, can possibly keep track of instructions that it has sequenced through in the past, but has no information about instructions that it is likely to be sequencing through shortly. If it can be informed about instructions that it is likely to encounter soon, it can perhaps make better sequencing and scheduling decisions. For example, in the Tera architecture, instructions contain a field which indicates how many future instructions are independent of the current instruction [1]. The compiler sets this field, and the hardware can make use of this information as it sees fit. Another example is the use of predecoded information that has been used in some recent microarchitectures. I expect to see more heavy use made of this type of information for a variety of purposes. This information could be conveyed explicitly, by software, as a part of the instruction stream, or could be determined dynamically and cached for future use.

2.2. Operation Execution and Speculation Support

Building logic blocks to execute operations is fairly routine, though there is still a lot of room for innovation in techniques to reduce operation *execution* latencies, perhaps by speculating on values of input operands. The more important issue is when to schedule an operation for execution. Worst-case assumptions delay the execution of operations, thereby degrading performance. Speculation is an important means of overcoming performance-limiting assumptions; regularities and patterns in programs allow speculation to be correct more often than not.

For many speculation has associated with it a notion of complexity. While it is certainly true that hardware would be simpler if no speculation were performed, the hardware to handle speculation is not necessarily very complicated; much of it is considered routine today. To make the reader feel comfortable about speculation, let me introduce the notion of *storage location speculation*. Here I have two storage structures, a small but fast structure, and a large but slow structure, where a named storage location might be present. A processor will speculate that the contents of the desired storage location reside in the faster storage, and if this speculation succeeds, the speed of access to the desired storage element will have been improved. Of course what I am talking about are caches, a form of speculation that has been ubiquitous in processor design for the last 30 years. The hardware to handle multiple storage location speculations simultaneously, i.e., a non-blocking cache, is more involved that hardware that allows only one speculation at a time (a blocking cache), but that is something that we know how to deal with today.

In the billion transistor era, I see heavy use of speculation of all forms: speculation will be applied whenever a performance-limiting situation occurs. Almost every modern processor makes use of *control speculation* to overcome control dependences. More recent approaches have proposed the use of *data dependence speculation* to overcome ambiguous data dependences [4], and *value speculation* to overcome true data dependences [3]. Other forms of speculation will be discovered as other performance-limiting situations arise. A point to keep in mind is that the hardware to support many forms of speculation is very similar, e.g., hardware for control speculation can also be (and is) used for recovery of a mispredicted data value.

An important consideration in deciding how much speculation to use is power consumption. More incorrect speculation means more wasted power. Rather than continue with incorrect speculation and expend power with no performance gain, it might be better to constrain the speculation. This is another reason why it will be important to have microarchitectures that allow for varying power-performance execution.

A concept that might reduce the need for speculation is *instruction reuse* [5]. Recent work has suggested that a large number of operations are re-executed with the same operands. In such cases, if we can determine what the outcome of an operation will be, without having to execute it, there is no need to speculate on the outcome, and verify the speculation by executing the operation. If branches could be reused, the need to predict their outcome diminishes. Likewise, if operation outcomes can be determined ahead of time, the need for value prediction is diminished. Success at instruction reuse (both in hardware structures to capture the phenomenon, and is software transformations to increase its occurrence) might cause us to rethink the need for aggressive speculation structures. Resources that would otherwise be spent in making more powerful speculation structures might be better spent in structures that improve instruction reuse, backed up by less powerful speculation structures.

2.3. Inter-Operation Communication and Memory Systems

The memory system has perhaps been the centerpiece of computer architecture studies; many researchers have suggested that it is fruitless to work on processors since the real problem is the memory system. In my opinion, such reasoning is flawed. The role of the memory system is to be able to support a desired rate of inter-operation communication during processing. The demands that are made of this resource depend upon how processing is carried out -- the memory system cannot be studied in isolation.

The memory system is a pool of storage resources that have the appearance of a single set of resources (thereby maintaining the image of named storage inherent in the program), but are physically implemented as different sets of storage resources (e.g., read/write buffers, multiple levels of caches, main memory, etc.), that attempt to match the latency and bandwidth demands of processing. With billion transistor chips, I expect that many of the resources will be devoted to storage structures that will serve as memory latency reducers and memory bandwidth amplifiers. In addition to the traditional organizations for storage, I also expect to see a pool of storage resources organized in a "value-oriented" fashion, akin to a token store in a dataflow machine. Let me draw an analogy to registers. Programs are written with the view that inter-operation register communication takes place through a set of architectural registers. However, modern superscalar processors have a physical register file that is larger than the architectural register name space. Much of the inter-operation communication takes place in this physical register storage space rather than the architectural register storage space. Moreover, many operations bypass the register file, instead being serviced from reservation station, bypass paths, etc. I see similar things happening with memory. Perhaps we will have storage structures that match producers (stores) and consumers (loads), allowing such communication to bypass the traditional storage structures of a memory hierarchy.

Where does processor and integrated memory fit in into the picture? Clearly billion transistor "chips" will include a fair amount of DRAM storage, in an attempt to have a system on a chip. Whether the logic and DRAM will be on the same die, or in separate die (but the same package) is not clear. If on the same die, I believe that the storage will be used to develop novel memory hierarchy components, such as above, rather than continue to maintain the traditional architectural appearance of processor and memory.

3. Software

A big change I expect to see in the billion transistor era will be in the role of software, especially compilers. Current compilers expend considerable effort in program analysis of all types, with a goal of giving guarantees, e.g., guarantees that 2 memory operations are independent. If the hardware has ability to speculate and correct itself, such guarantees are not needed. A compiler could analyze a program to suggest to the hardware that parts of it might be executable in parallel, as opposed to guaranteeing to the hardware that they can be executed in parallel. In particular, if the hardware has the ability to speculate on data dependences, there might be little reason to guarantee that two operations are indeed independent. Relieving the compiler from having to make such decisions might simplify the analysis required, resulting in simpler compilers. I expect the pendulum to swing from the mindset of simple hardware supported by guarantees from the compiler (as was characterized by the RISC era) to one of smart hardware supported by compilers that help the hardware to the extent possible, but are not asked to make guarantees, i.e., smart hardware and simple compilers. Ways in which compilers could assist the hardware might include methods to improve the (run-time) predictability of branches and values.

4. Algorithms

I believe that billion transistor chips will cause us to revisit algorithms and data structures. To date work in algorithms has been in two camps: (i) sequential algorithms and (ii) parallel algorithms, or more accurately large-scale parallelism. In the former case, the objective is to minimize the number of operations executed, possibly at the expense of aggravating the critical path through the computation -- nothing is to be gained by reducing the critical path if operations are going to be executed one at a time. In the latter case, the algorithms are targeted for large-scale parallel machines. These algorithms reduce the critical path through the computation, but execute many more operations than a sequential version. Billion transistor chips will be "small-scale parallel". Here both the critical path through select portions of the computation (the instruction window) as well as operation counts are important. In addition, the sizes and organizations of the chosen data structures impact performance. I expect to see the development of small-scale parallel algorithms, possibly customized for the chosen microarchitectures.

5. Conclusions

The next 15 years promise to be even more exciting that the past 15 years, as we move to the era of computing with billion transistor chips. I expect to see considerable innovation in microarchitectures that allow us to make productive use of a billion transistors in solving the computing problem. I expect that these microarchitectures will change the role of the compiler from one of giving guarantees to one of providing assists. I also expect to see considerable innovation in algorithms and data structures that are tailored for the small-scale parallel model of billion transistor chips.

Acknowledgments

Many of the ideas and opinions presented in this paper are a result of discussions with current and former students: Todd Austin, Steve Bennett, Scott Breach, Manoj Franklin, Andy Glew, Andreas Moshovos, Dionisios Pnevmatikatos, Avinash Sodani, and T. N. Vijaykumar. My thanks to them all. I would also like to thanks sponsors of our research: DARPA, Intel, NSF, and ONR.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, ‘‘The Tera Computer System,’’ in *Proc. Int. Conference on Supercomputing*, Amsterdam, pp. 1-6, June 1990.

[2] M. S. Lam and R. Wilson, ‘‘Limits of Control Flow on Parallelism,’’ in *Proc. 19th Annual Symposium on Computer Architecture*, Queensland, Australia, pp. 46-57, May 1992.

[3] M. H. Lipasti and J. P. Shen, ‘‘Exceeding the Dataflow Limit via Value Prediction,’’ *Proc. MICRO-29*, December 1996.

[4] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, ‘‘Dynamic Speculation and Synchronization of Data Dependences,’’ *Proc. 24th Annual Symposium on Computer Architecture*, June 1997.

[5] A. Sodani and G. S. Sohi, ‘‘Dynamic Instruction Reuse,’’ *Proc. 24th Annual Symposium on Computer Architecture*, June 1997.

[6] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, ‘‘Multiscalar Processors,’’ in *Proc. 22th Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.