How Low Can You Go?

Henry Dietz

University of Kentucky, Lexington KY 40506, USA,
hankd@engr.uky.edu,
WWW home page: http://aggregate.org/hankd

Abstract. It could be said that much of the evolution of computers has been the quest to make use of the exponentially-growing amount of on-chip circuitry that Moore predicted in 1965 – a trend that many now claim is coming to an end[1]. Whether that rate slows or not, it is no longer the driver; there is already more circuitry than can be continuously powered. The immediate future of parallel language and compiler technology should be less about finding and using parallelism and more about maximizing the return on investment of power.
Programming language constructs generally operate on data words, and so does most compiler analysis and transformation. However, individual word-level operations often harbor pointless, yet power hungry, lower-level operations. This paper suggests that parallel compilers should not only be identifying and manipulating massive parallelism, but that the analysis and transformations should go all the way down to the bit or gate level with the goal of maximizing parallel throughput per unit of power consumed. Several different ways in which compiler analysis can go lower than word-level are discussed.

Keywords: precision, accuracy, bit-slice, logic optimization

1 A Word About Words

Throughout the history of computers, programming systems have taken a multitude of different approaches: procedural, declarative, functional, ... However, in nearly all cases, typed data objects are treated as indivisible entities. A `REAL` in Fortran is a thing; it may be operated upon, but whatever happens to it happens to it as a whole unit. In lower-level languages like C, machine words can be dressed as abstract data types, but they retain all the properties of machine words. This paper suggests it is time for compiler technology to start looking inside basic word-level data and operations.

There are two different ways that sub-word analysis can be approached. The following section discusses methods by which words may be segmented into smaller words or fields to avoid processing meaningless bits. The section following that describes full bit-level analysis and optimization to remove unnecessary power use at the gate level.

2 Not All The Bits, Not All The Time

In languages and compilers for parallel computing, the focus has largely been on utilizing as much parallelism as the hardware provides, with the expectation that the maximum speedup will be obtained. At Thinking Machines in the 1980s, the mantra was "all the wires all the time" – and this notion of keeping all the hardware busy all the time certainly predates the 1980s and has persisted. However, it is now practical to have far more circuitry on a chip than can be continuously powered. The implication is that languages and compilers must become more careful about not wasting power on unnecessary computation.

Whenever a programmer writes code manipulating numeric data, they will likely consider if the values are known to always be integers, and most often will declare the variable as `int` if so, and `double` otherwise. Even the `unsigned int` type is rarely used, which is strange given that array index values are normally non-negative. Accepting these sloppy type declarations implies extra work – and extra power consumption – for processing what are essentially useless bits.

2.1 Integer Range Analysis

In most modern programming languages, an integer is implemented by hardware operating on a word with at least 32 bits. In code you have written, does each integer really have values spanning the complete range of 32-bit integer values, from -2,147,483,648 to 2,147,483,647? Of course not.

For example, if an integer `int i;` is used to index an array declared as `double a[1000];`, it is fairly clear that `i` should be in the range 0..999. That range spans fewer than 1024 values, which means it fits within 10 bits. So, what happens to the other 22 bits of the 32-bit integer? The answer is everything – even though the result is to store the same 22-bit zero value that was already there into `i`'s top bits. If the 10-bit value was treated as a 16-bit value, instead of 32-bit, less than half as much ALU and datapath circuitry would *need* to be active.

Originally, the C programming language did not specify how many bits were used to represent an `int` – it merely stated that an `int` was "an integer, typically reflecting the natural size of integers on the host machine" and listed as examples 16 bits for a DEC PDP-11, 36 bits for Honeywell 6000, and 32 bits for IBM 370 and Interdata 8/32[2]. However, this made many programs non-portable between machines. To improve portability, programmers used macros to define portable names for declaring types of various common sizes, which eventually led to standards like `int32_t` meaning a 32-bit integer in C++. Ironically, the original C language includes a fully general syntax for specifying the exact number of bits in an integer, but the syntax was only allowed for fields of a `struct`. For example, `struct { int:5 a; unsigned:1 b; } ab;` specifies that `ab.a` is a 5-bit signed integer and `ab.b` is 1-bit unsigned integer.

Certainly, it makes sense for programming languages to allow specification of the number of bits in each value. However, there is even ambiguity in such a specification: does `int:5` mean "5-bit integer" or "integer of at least 5 bits"? As bitfield specifications, C treated the specification as the exact number of bits rather than a minimum, but arguably it would often be useful, for example, to store a 5-bit integer in an 8-bit byte to meet memory access alignment constraints.

From a compilation point of view, the key is not just tracking the size of data objects, but using analysis to determine the set of active bits. Thus, the loop index variable declared as `int i;` could be automatically transformed by compiler analysis into something like `unsigned i:10;`. The necessary compiler value range analysis has long been known and used to perform loop unrolling/unraveling, improve efficiency of array bounds checking, and support dependence analysis... and it was used to infer types of variables as early as the mid-1960s[3]! The suggestion of the current work is simply that this type of analysis of value range be used both to adjust the declared size and type (e.g., signed vs. unsigned), and also to restrict operations to cover the active bits within that value's storage representation. Reusing the loop index example, even if `i` is stored as `int i:32;`, it is perfectly valid to operate only on the bottom 16 (or 10) bits in code where the top bits are known to be unaffected.

2.2 Floating-Point Accuracy, Not Precision

While tracking the size of integers is straightforward, the logically equivalent analysis for floating-point values is significantly more complex.

Operations on floating-point representations are inherently imprecise. The whole concept of floating-point is based on using an approximate representation of values to allow greater dynamic range with fewer bits. New hardware supports several floating-point precisions with huge performance benefits in use of lower precisions while remaining in the same power budget. Peak performance of the AMD RADEON INSTINCT MI25 GPU[4] is 768 GFLOPS 64-bit, 12.3 TFLOPS 32-bit, and 24.6 TFLOPS 16-bit – a factor of 32X faster using the lowest precision instead of 64-bit `double`.

According to IEEE 754[5], the larger exponent field of 64-bit `double` gives it greater dynamic range than a 32-bit `float`, but lack of dynamic range is rarely why `double` is specified. The 24-bit mantissa of a `float` is accurate enough to express nearly any physical measurement for input or output. However, it is difficult to analyze the accuracy resulting from errors compounded while performing an arbitrary computation; the 53-bit mantissa of a `double` is treated as having so many *guard bits* that the programmer is comfortable accepting the results as accurate. This is a false security; not only is efficiency sacrificed, but when `float` is insufficient, `double` often also fails to deliver accurate results[6].

Rather than requiring the programmer to specify the precision, perhaps needed accuracy should be specified, and the compiler tasked with picking the appropriate precision. Unfortunately, compile-time analysis of accuracy bounds is inherently conservative, disturbingly often revealing that even `double` arithmetic is insufficient to guarantee an acceptably accurate result for worst-case input data, although lower precisions might suffice for every dataset actually used. Most programming languages also lack syntax for specifying accuracy. Both these problems can be resolved by adding a language construct (or pragma) that allows a user-specified accuracy acceptance test[6], which can be used to speculatively perform each computation at the minimum feasible precision, automatically repeating the computation at higher precision if necessary.

There is yet another advantage to specifying accuracy rather than precision for floating-point values: the values do not need to be represented using floating-point at all! For example, it may be advantageous to use LNS (logarithmic number system)[7] or to map the computation into scaled integer arithmetic. IEEE floating-point formats are primarily portable data exchange representations.

2.3 Smaller Data Fits

A consequence of the transformations discussed above for both integer and floating-point data is that lower precision values can fit in a smaller memory footprint. That principle holds true whether the memory is DRAM used for main memory or SRAM registers in a processing element. Reducing memory footprint reduces power by implying transmission of fewer bits, and can result in far greater power savings if the smaller data can more often reside in a higher level of the memory hierarchy.

Originally, the concept of SWAR (SIMD within a register)[8] was primarily to obtain modest speedups by packing a vector of smaller data objects into each register and performing SIMD-parallel operations on the packed data. Nearly all processors provide support for such packing, most in the form of SWAR instructions as seen in Intel AVX[9] or ARM NEON[10], but also in the form of "Advanced Vector Extensions" in RISC-V[11]. Operations on packed data significantly reduce power consumption while facilitating parallel execution.

For non-SIMD non-vector code, this packing requires compiler tracking of the occupancy of portions of registers, not just complete registers. A 64-bit register might hold four 16-bit values, and the lifetimes of those values may be different; thus, it is possible that a fraction of the register might be free for reuse while other parts are occupied by live values. Early work attempting to create SWAR register packings was called common subexpression induction[12], and the tracking of partial liveness was mentioned, but the analysis was never fully developed. Perhaps it is now time? There is no need to restrict layout analysis to registers; similar benefits can be obtained packing cache lines.

3 From Bits To Words, And Back Again

In 1958, the vacuum-tube-based EDSAC 2 computer[13] used the innovative trick of bit-slicing: implementing word-level instructions by executing microcoded sequences of operations on a smaller number of bits at a time. This approach was very widely adopted throughout the 1970s; for example, the AMD Am2900-series bipolar logic chips[14] provided 4-bit slice components that were used in a wide variety of computers including various DEC PDP-11 models and the UCSD Pascal P-machine processor. Use of bit-slicing greatly simplified the hardware, lowering cost at the expense of serial execution speed. By the 1980s, circuitry was cheap enough for most computers to operate on full words at a time, with word size slowly increasing from 8, to 16, to 32, and finally to 64 bits.

Massively-parallel computers have largely followed the same pattern. Fewer gates per processing element meant more parallelism, hence greater speedup, so many parallel supercomputers reduced circuitry per processing element by slicing. The ICL Distributed Array Processor (DAP)[15], STARAN[16], Goodyear Massively Parallel Processor (MPP)[17], Thinking Machines CM and CM2[18], and NCR GAPP[19] sliced at the single-bit level; the somewhat-later MasPar MP-1[20] was built using 4-bit slices. As 32-bit microprocessors became cost effective, and especially with the birth of Linux PC cluster supercomputing in 1994, massively-parallel computers began to migrate to processing elements that operate on a word at a time. Now, even GPUs with thousands of processing elements on a single chip operate on at least 32-bit words.

The point is that word-parallel hardware was about speeding up serial operations on word data at the cost of higher circuit complexity per unit performance. For example, a 32-bit adder needs additional logic (e.g., implementing carry lookahead) to perform one 32-bit add as fast as 32 one-bit adders can each perform 1/32 of a 32-bit add. This difference has become critical in the power-limited world. As a result, I believe we are now in the early days of a rebirth in bit-level massively-parallel processing. This brings at least two new challenges for optimizing, parallelizing, compiler technology.

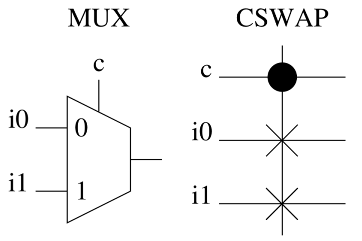
3.1 True Bit-Level Optimization

Mapping of word-level algorithms into optimized bit-level implementations must be addressed. Earlier bit-sliced systems generally did not do this; they would use a generic microcode subroutine to handle each word-level operator rather than optimizing at the bit level. An early example of true bit-level optimization is the BitC language and compiler[21]. For example, BitC generates 3,040 gate operations for an 8-bit multiply, but just 64 for 8-bit squaring.

The new bit-level targets are not just conventional gates, but include FPGAs and reconfigurable logic, adiabatic circuits, quantum logic, etc. For example, TrueNorth[22] is fundamentally a massively parallel bit-serial machine with a somewhat unusual gate structure; although there is some compiler infrastructure for mapping neural networks to it, there is no fundamental reason why parallel algorithms in general could not be transformed to target it (in fact, watching the TrueNorth videos, I have flashbacks of Danny Hillis talking about the interconnections between processors in the connection machine[18]).

3.2 Whole Program Scale Gate Optimization

Thus far, very little optimizing compiler work has attacked the problem of optimizing programs at what might be called the circuit model level. There are lots of gate-level circuit optimization tools, especially for SOP (sum of products) form using AND, OR, and NOT gates with arbitrary many inputs (e.g., Espresso[23]), but these tools are ill-suited to processing logic representing complete programs.



c	i0	i1	MUX	CSWAP
0	0	0	0	0 0 0
0	0	1	0	0 0 1
0	1	0	1	0 1 0
0	1	1	1	0 1 1
1	0	0	0	1 0 0
1	0	1	1	1 1 0
1	1	0	0	1 0 1
1	1	1	1	1 1 1

Fig. 1. MUX and CSWAP (Fredkin) Gates And Logic Functions

Optimizing more realistic hardware designs also using XOR gates, or using only a single, fixed number of inputs, universal gate such as NAND, NOR, or MUX (1-of-2 multiplexor), is also an unsolved problem. Genetic algorithms and other machine learning techniques can help.

In addition, a variety of new types of gates have been created for adiabatic and quantum computation. One of the more promising is the Fredkin, or CSWAP (conditional swap), gate. As Figure 1 shows, CSWAP is essentially a MUX with an extra output, so it is obviously universal, but has the interesting property that the number of 1 bits entering and leaving is preserved, making it particularly suitable for adiabatic (thermodynamically reversible, very low power) implementation. A quantum CSWAP implementation was reported last year[24]. Thus, converting programs into CSWAP logic could be a path to very low power consumption and high performance. A key complication is that CSWAP gates do not allow fanout; minimizing the number of *ancilla* "garbage" wires added to copy signals while conserving the number of 1s is a non-trivial design problem.

4 Conclusion

Parallel languages and compilers had mostly been about the large. This paper suggests it is time to be looking at very large numbers of very small, low-level, details – many intricately intertwining compiler technology and architecture.

We have been slowly moving in the directions discussed in this paper for over two decades[6][12][21]. Earlier this year, we made a significant advance toward targeting various bit-level architectures, especially those involving adiabatic or quantum logic: a compiler that converts a program written in a subset of C directly into a full custom gate-level hardware design[25]. This compiler first performs conventional analysis and various optimizations, then inserts explicit manipulation of a state variable to implement control flow. Each word-level value is then decomposed into a vector of bit-level operation DAGs computing the individual bits, and the bit-level code for all basic blocks in the program is optimized as a single combinatorial logic circuit. The resulting circuit can be output in any of a variety of forms, including as a combinatorial Verilog module that implements the C code when fed a series of clock pulses. Of course, this compiler is just a crude proof of concept; there is still a long way to go.

References

1. Fletcher, S.: Computing after Moore's Law. Scientific American, https://www.scientificamerican.com/article/moores-law-computing-after-moores-law/ (May 1, 2015)
2. Kernighan, Brian W. and Ritchie, Dennis M.: The C Programming Language. Prentice Hall, ISBN 0-13-110163-3 (1978)
3. Klerer, M. and May, J.: Two-dimensional Programming. Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I, 63–75 (1965)
4. AMD: RADEON INSTINCT MI25. http://instinct.radeon.com/_downloads/radeon-instinct-mi25-datasheet-15.6.17.pdf (accessed July 2017)
5. IEEE, IEEE Standard for Binary Floating Point Arithmetic Std 754-1985. (1985)
6. Dietz, H., Dieter, B., Fisher, R., and Chang, K.: Floating-Point Computation with Just Enough Accuracy. Lecture Notes in Computer Science, Volume 3991/2006, ISSN: 0302-9743, pp. 226–233 (2006)
7. Chugh, M. and Parhami, B.: Logarithmic arithmetic as an alternative to floating-point: A review. 2013 Asilomar Conference on Signals, Systems and Computers, 1139–1143 (2013)
8. Fisher, R. and Dietz, H.: Compiling for SIMD Within a Register. Languages and Compilers for Parallel Computing: 11th International Workshop, LCPC'98, Springer, ISBN 978-3-540-48319-9, 290–305 (1999).
9. Lento, G.: Optimizing Performance with Intel Advanced Vector Extensions. Intel White Paper (September 2014)
10. ARM: NEON. https://developer.arm.com/technologies/neon (accessed July 2017)
11. Waterman, A. and Asanovic, K. (editors): The RISC-V Instruction Set Manual, Volume 1: User-Level ISA, Document Version 2.2. RISC-V Foundation (May 2017).
12. Dietz, H.: Common Subexpression Induction. Technical Report TR-EE 92-5, School of Electrical Engineering, Purdue University (January 1992)
13. Wilkes, M. V.: EDSAC 2. IEEE Annals of the History of Computing, Volume 14, Issue 4, ISSN 1058-6180, 49–56 (1992)
14. Advanced Micro Devices: The Am2900 Family Data Book With Related Support Circuits. Advanced Micro Devices (1979)
15. Reddaway, S. F.: DAP - a distributed array processor. Proceedings of the 1st annual symposium on Computer Architecture, ACM Press, 61–65 (1973)
16. Batcher, K. E.: STARAN parallel processor system hardware. National Computer Conference, pp. 405-410 (1974)
17. Batcher, K.: Design of a Massively Parallel Processor. IEEE Transactions on Computers, Volume C-29, Issue 9, 836–840, (September 1980)
18. Tucker, L. W. and Robertson, G. G.: Architecture and applications of the Connection Machine. IEEE Computer, Volume 21, Number 8, 26–38 (August 1988)
19. Morely, R. E. and Sullivan, T. J.: A massively parallel systolic array processor system. Proceedings of the International Conference on Systolic Arrays, 217–225 (1988)
20. Blank, T.: The MasPar MP-1 architecture. Thirty-Fifth IEEE Computer Society International Conference, Compcon, 20–24 (1990)
21. Dietz, H. G., Arcot, S. D., and Gorantla, S.: Much Ado about Almost Nothing: Compilation for Nanocontrollers. Languages and Compilers for Parallel Computing (LCPC 2003), Springer, ISBN 978-3-540-24644-2, 466–480 (2004)
22. Sawada, J., et al.: TrueNorth Ecosystem for Brain-Inspired Computing: Scalable Systems, Software, and Applications. IEEE/ACM SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 130–141 (2016)

23. Brayton, R. K., Sangiovanni-Vincentelli, A. L., McMullen, C. T., and Hachtel, G. D.: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, ISBN 0898381649 (1984)
24. Patel, R. B., Ho, J., Ferreyrol, F., Ralph, T. C., and Pryde, G. J.: A quantum Fredkin gate. Science Advances, Volume 2, Number 3 (March 25, 2016)
25. Dietz, H. G.: Spring 2017 EE599-006/EE699-007 Optimizing Compilers, “Hardly Software” Class Project. Electrical and Computer Engineering Department, University of Kentucky (May 5, 2017)