Parallel Bit Pattern Computing

Henry G. Dietz
Electrical and Computer Engineering
University of Kentucky
Lexington, KY USA
hankd@engr.uky.edu

***Abstract*—There are many ways to reduce power consumed in performing a computation. Most focus on making each gate more power efficient. In contrast, the current work focuses on directly reducing the number of gate-level operations needed to produce each word-level result.**

Compiler optimization of computations at the gate level exposes many redundancies that are not apparent when optimizing word-level operations. In the proposed architecture, all operations on multi-bit data values are performed bit serially. Thus, a k-bit add takes $O(k)$ clock cycles. However, by doing each operation SIMD-parallel on n data, n k-bit operations also complete in $O(k)$ clock cycles using only $O(n)$ gates per clock. Further improvement can be made by using regular expression patterns to represent the n values in each bit position; not only does this compress the data, but it also allows many gate-level operations to be performed directly on the patterns without expanding them to bit vectors.

***Index Terms*—low power; green computer architecture; compiler optimization; gate-level logic optimization; regular expression; just in time compilation**

I. Introduction

For the last half century, the majority of improvement in the abilities of computers has come from using more circuitry to perform parts of the computation in parallel. Over that time, there was steady exponential growth in the amount of circuitry that can be placed on each chip, but the growth rate predicted by Moore in 1965 is no longer being met[1]. As that rate slowed, a second problem developed: it became impractical to continuously power all the circuitry that can fit on a chip. Thus, the next big improvements in system performance will not come by simply using more circuitry in parallel. Power per unit of computation also must be reduced.

There are many different ways in which power consumed per unit of computation performed can be reduced[2]. In the current work, we suggest a combination of compiler optimization technology and computer architecture that can productively apply all of the methods described in the following subsections.

A. Implement Using Low-Power Gates

An obvious way to reduce power consumption is to employ gate designs which inherently use less power. Adiabatic logic[3] can significantly reduce power consumption per active gate. Transformation of gate-level designs into reversible logic[4] can be applied to reduce power consumption of nearly any computer architecture.

B. Operate Only On Active Bits

Most programming languages are word-oriented: the data types in the language roughly correspond to machine words. For example, the C langauge originally defined an `int` as being an integer value of whatever precision was most efficiently manipulated by the machine – now, an `int` is typically a 32-bit word value. The catch is that fetching, operating on, and storing 32 bits often implies expending power processing bits that are known to be inactive and unnecessary.

Operations on integers should be limited to just the active bits. A variable that is used as a loop index running from 0 to 100 should not be treated as a 32-bit value. In SWARC[5] or BitC[6] one can declare such a variable as `unsigned int:7`. The SWARC compiler minimizes operations on inactive bits by packing shorter values into machine words that are operated upon as "SIMD Within A Register" (SWAR) vectors. BitC targets bit-serial nanocontrollers that directly implement arbitrary-precision arithmetic.

Conceptually the same type of precision adjustment can be applied to the precision of floating-point values and arithmetic. For example, specifying accuracy constraints rather than precision allows the compiler to use higher precisions only when lower precision fails to meet accuracy requirements[7].

C. Apply Compiler Optimization At The Gate-Level

Optimizing compilers can perform a large variety of sophisticated transformations that simplify computations and remove unnecessary operations, thus saving 100% of the power those operations would have consumed.

For example, constant folding, common subexpression elimination, and value forwarding allow code like `a=4; b=a-3; c=c+b; d=a*c; e=c*a*b;` to be optimized into `a=4; b=1; ++c; d=c<<2; e=d;`. More sophisticated compiler analysis and transformations can even change the order of complexity of a computation by eliminating entire loops, etc. The perhaps surprising fact is that the same types of analysis and transformation compilers normally apply to word-level operations becomes dramatically more effective when applied to bit-level representations of computations[6][8]. Ironically, bit-serial SIMD supercomputers generally have not taken advantage of this; from the Goodyear MPP[9] to the CM-2[10], they instead have been programmed by invoking bit-level microcoded instruction sequences that implement a relatively conventional word-oriented instruction set.

978-1-7281-5416-9/19 $31.00

Consider bit-serial addition of `c=a+b` where `a` and `b` are declared as `unsigned int:4 a, b;`. The basic gate-level operations on individual bits would be the following 35 single-gate operations:

```
c0=(a0^b0); c1=((a0&b0)^(a1^b1));
c2=(((a1&b1)|((a0&b0)&(a1^b1)))^
    (a2^b2));
c3=(((a2&b2)|(((a1&b1)|((a0&b0)&
    (a1^b1)))&(a2^b2)))^(a3^b3));
c4=((a3&b3)|(((a2&b2)|(((a1&b1)|
    ((a0&b0)&(a1^b1)))&(a2^b2)))&
    (a3^b3)));
```

However, ordinary compiler optimization would factor-out the repeated portions of the computation. The result is just 17 single-gate operations and about half the power usage:

```
c0=(a0^b0); t0=(a0&b0); t1=(a1^b1);
c1=(t0^t1); t2=(a1&b1); t3=(t0&t1);
t4=(t2|t3); t5=(a2^b2); c2=(t4^t5);
t6=(a2&b2); t7=(t4&t5); t8=(t6|t7);
t9=(a3^b3); c3=(t8^t9);
c4=((a3&b3)|(t8&t9));
```

Taking this analysis a bit further, if the addition is preceded by `b=1;`, the compiler analysis produces the equivalent of an incrementer circuit instead of an adder. This uses just 7 single-gate operations:

```
c0=~a0; c1=(a0^a1); t0=(a0&a1);
c2=(a2^t0); t1=(a2&t0); c3=(a3^t1);
c4=(a3&t1);
```

If instead `b=a;` appeared before the addition, then the result becomes the zero-gate shift computation:

```
c0=0; c1=a0; c2=a1; c3=a2; c4=a3;
```

Although the improvement obtained varies widely, some complex computations can realize as much as a five-order-of-magnitude reduction in the number of gate-level operations needed as compared to when optimization is done exclusively at the word level[2].

D. Amortize Control Logic Overhead

In thinking about computation, there is a tendency to focus on the arithmetic and memory accesses as the primary costs. Those costs are significant, but the basic circuit and power overhead of fetching, decoding, and implementing the control logic for executing instructions easily can be comparable or even higher. It is critical that more clever control logic be implemented in a way that does not exceed the logic power savings by making more control circuitry active.

The unfortunate fact is that more sophisticated, finer-grained, control of a computer requires a significantly larger investment of circuitry and power budget for the control logic. Consider a bit-serial machine with a single-gate one-bit ALU. It can be made to do only the necessary bit-level operations, and easily can be made to draw very little power for the ALU. However, the power invested in fetching and decoding an instruction will be far greater than that consumed by the ALU. Just incrementing a program counter to point at the next instruction uses far more power than performing a single-gate ALU operation.

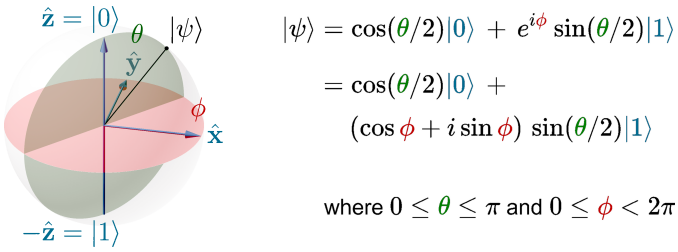

Fig. 1. Bloch Sphere visualization of a *qubit* with value ψ

A popular solution to this is to use "virtualized" SIMD-parallel execution to hide latency: a model very similar to what was used for the Thinking Machines supercomputers[10], and more recently for GPUs[11]. If incrementing the program counter, fetching an instruction, and decoding the operation costs approximately N gate delays, then there should be a virtualization factor of at least N. For example, if instruction control logic takes about 20 gate delays to handle one instruction, then a sequence of approximately 20 single-gate operations should be processed per instruction. This virtualization factor is, of course, multiplied by the width of the parallel system. A machine with 32 single-gate ALUs might therefor be virtualized with at least 640 SIMD processing elements.

E. N-Way Parallel Execution Without N Units of Hardware

One of the most promising potential methods for dramatically reducing power consumption per unit computation is the use of an execution model that supports N-way parallel computation with fewer than N units of work. There are various ways in which this apparently impossible goal might be achieved. The current work is not about quantum computing, but the proposed approach can be better understood by considering how a quantum machine might achieve this goal.

A quantum bit (aka, a *qubit* or *qbit*) can represent the value 0, 1, or a *superposition* of both simultaneously. Physicists describe the value of a qubit as a continuous wave function that defines probabilities of the value being 0 or 1. A *Bloch Sphere*, as shown in Figure 1, is a geometrical representation of this wave function value, essentially defining the probabilities of 0 and 1 in terms of a pair of spherical coordinates (θ, ϕ) defining a position on the surface of the sphere, with 100% 0 at the top of the sphere and 100% 1 at the bottom.

The interesting feature of this wave function model is that multiple qubits can be *entangled*, made to have their wave functions interact, such that any set of combinations of n discrete 0 and 1 values are represented using just n qubits. For example, two qubits can hold 00, 01, 10, and 11 simultaneously, with each of those possible discrete values having its own probability. In such a configuration, any operation

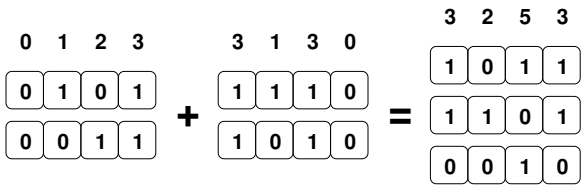

Fig. 2. Addition of two 2-pbit values

performed on the qubits is essentially performed on all values simultaneously – by changing the wave function in a way that can modify the probability distribution. It is this ability to represent up to 2^n values in just n qubits, and to have performing a single operation act upon 2^n values, which makes quantum computation so desirable.

Those abilities of quantum computing come at a cost: what a physicist would call a *superposition state* does not actually exist as what a computer engineer would call a *state*. *Quantum coherence* is maintained only for a very limited time and even within that period the act of reading-out an *n-qubit* value essentially collapses the superposed state to a randomly-selected discrete value. Thus, you can compute many results simultaneously, but can only read one. There is also the fact that nobody has yet built a quantum computer that is large and reliable enough to do useful computations faster than a conventional computer.

Instead of building a quantum computer, the current work begins with the notion of simulating quantum-like superposition and entanglement using *bit patterns* implemented in a relatively conventional computer. Each *pattern bit* (abbreviated as *pabit* or *pbit*) is an ordered list of 1-bit values. The entanglement of two pbits is represented by pairing of bit values in corresponding positions within their ordered lists. For example, two pbits with the value $\{0, 1, 2, 3\}$ and two pbits with the value $\{3, 1, 3, 0\}$ represents the entangled set of value-pairs: $\{(0,3), (1,1), (2,3), (3,0)\}$. As shown in Figure 2, adding these two produces the 3-pbit result $\{3, 2, 5, 3\}$. Clearly, the ordered pattern for each bit can be treated as a (in this case, 4-bit) vector and bit-serial operations on pbits can be performed using SIMD-parallel gate-level operations.

In this representation, the length of each bit vector is determined by its entanglement and can be quite large: n-way entanglement requires a bit vector of length 2^n. However, rather than directly representing the bit vector, the pattern of bits can be compressed by representing the bit vector as a *regular grammar* or *regular expression* (*RE*) that can generate the bit vector. The problem of finding the minimal complexity RE to generate a given bit pattern is difficult, but *run-length encoding* (*RLE*) is easily applied to create an RE that can deliver significant compression. For example, the bit vector $\{0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1\}$, trivially can be compressed into a pattern like $0^6 1^4 0^4 1^2$ by RLE.

To implement the SIMD virtualization discussed earlier (as a method to reduce control overhead), the RE can use a symbol size of greater than one bit: the vectors can be

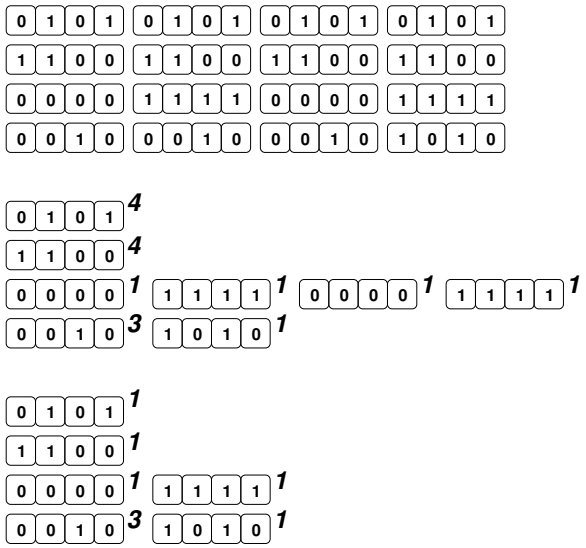

Fig. 3. Encodings of 4 fully entangled pbits using 4-bit vector chunks

"chunked" into sub-vectors of length corresponding to the desired virtualization times the physical parallelism of the hardware. In practice, chunks should be fairly large; the current implementation, which runs on a conventional laptop, uses a chunk size of 2^{12} bits. For a small example, consider just 2^2 bits per chunk. The 4-pbit *pattern integer* (*paint* or *pint*) with the 4-way entangled value $\{2, 3, 8, 1, 6, 7, 12, 5, 2, 3, 8, 1, 14, 7, 12, 5\}$ could be segmented into 4-bit-long symbols (chunks) and represented using RLE-generated RE as shown in Figure 3. As is also shown in that figure, it is easy to reduce the RE further by recognizing that the first three pbits are repeating patterns with lower entanglements.

While this RE compression scheme reduces the memory space needed, the key benefit is that **by operating directly on the compressed form, the amount of parallel computation needed can be reduced by as much as the compression factor**. While the reduction in the number of gate-level operations that must be performed depends on the entangled values and is not always exponential, it is generally a very significant factor. In the current implementation, duplicate chunks are also recognized and factored, further reducing storage requirements and allowing for *applicative caching* to avoid chunk-level recomputation. For example, if two chunks have been bitwise-ANDed before, that can be recognized and the result can be obtained from a software cache without ever accessing the data bits from the chunks.

II. The Prototype Implementation

The current prototype implementation consists of a little over 2K lines (40KB) of C source code. There are expected to be five major layers in the implementation of this model, four of which are operational at this writing. The lowest level is the chunk management. Above that layer is the factored bit parallel (FBP) layer, which manages regular expression values of pbits. The pbit layer is next, constructing optimized DAGs

Operation Name	Operation Functionality	O( )
H, Hadamard	a=superposition of 0/1	R
NOT	Adiabatic a =NOT a	N
SWAP	Adiabatic exchange values of a and b	1
CCNOT, Toffoli	Adiabatic if a AND b, c =NOT c	N
CSWAP, Fredkin	Adiabatic if c, SWAP(a, b)	N
Duplicate	$a = b$	R
AND	$a = b$ AND c	N
OR	$a = b$ OR c	N
XOR	$a = b$ XOR c	N
All	Reduction, true if a is 1	1
Any	Reduction, true if a contains a 1	1
Population	Reduction, count of 1s in a	R
Simplify	Internal simplify regular expression	R

TABLE I
WORST-CASE COMPLEXITY OF SOME FBP OPERATIONS

(directed acyclic graphs) for pbit-level computations. The pint layer handles arithmetic and other operations on entire multi-pbit signed and unsigned integer values. The top layer, which is not yet complete, will essentially wrap the pint layer in C++ constructs that allow `pints` to be directly manipulated in a C++ program as though they were a built-in data type.

How expensive are operations on pbits? Thanks to the chunk and FBP layers, many operations are surprisingly cheap. Table I lists some of the currently implemented FBP operations. For each, the operator name, functionality, and worst-case computational complexity are listed. Complexities are given in terms of N being the number of machine bits in the representation of the value, although it is highly unlikely that the regular expression representation will require examining that many bits; most patterns contain repeated symbols, and the bits in repeated symbols (chunks) are generally examined only once. A few operations have complexity bounded by the number of symbols in the regular expression, R. The O(1) complexities are achieved by algorithms that literally never examine an actual chunk, nor even walk the regular expression. Note that the reduction operations are logically equivalent to using interference in a quantum computer to sample a superposed state without collapsing it; of course, unlike qubits, pbit values can always be read without collapsing them.

Quantum computing compilation and/or simulation environments generally define, and expose to users, some simple syntax for expressing operations on *qubits*: a "quantum assembly language." For example, Quil[12], OpenQASM[13], and cQASM[14] all implement similar syntax for specifying operations on qubits. In contrast, the pbit layer here is really a just-in-time optimizing compiler for pbit operations. Various algebraic simplifications are symbolically performed as pbit expression DAGs are created, without any FBP activity. For example, AND of anything with the constant 1 does not create an AND gate, but returns a reference to the other operand. It is worth noting that the only constants that can appear in pbit DAGs are 0, 1, and Hadamard superpositions for anywhere from 1 to 32 entangled pbits.

High-level languages for quantum computers generally require specifying individual operations on qubits. For example, both IBM's Qiskit[15] and Microsoft's Q#[16] essentially add a variety of functions to existing languages to allow qubit-level specification of computations. In contrast, the system described in the current work also augments a conventional language (C/C++), but it directly provides a layer that understands pint operations and implements a just-in-time optimizing compiler to create the pbit DAGs.

A pint is represented as a data structure which contains an array of pbit references, a precision, and a flag specifying if the value is signed (as opposed to unsigned). All the usual integer operations are supported for pint containing from 1 to 32 pbits. The integer operations are supported by implementing an optimized gate-level circuit design to produce the value for each pbit. Some multi-bit integer operations are trivially lowered to operations on individual bits. For example, bitwise AND of two pint values trivially produces a result using ANDs of corresponding component pbit values from the two operands. Other operations are significantly less straightforward. For example, addition of two pint values performs a sequence of pbit operations that is equivalent to implementing a ripple carry adder circuit. Multiply builds upon that to implement a purely combinatorial shift-and-add circuit.

In our system, when the pbit layer is initialized, all the layers below are also initialized. Operations on pint simply compile optimized DAGs for the component pbit operations; the FBP layer is not triggered until the conventional value(s) of a pint are requested ("measured"). At that point, the pbit DAGs for the requested pint value are walked and decorated with references to the results of FBP evaluation of each pbit. Arbitrarily complex intermediate steps combining pint values do not cause any computation until it is demanded by calling for evaluation of a particular pint.

As a simple example, consider the problem of finding the square root of a 16-bit number, 29929 in this example. A complete C program to perform this pint computation is:

```
int main(int argc, char **argv) {
  pbit_init();
  pint a = pint_mk(16, 29929);
  pint b = pint_h(8, 0xff);
  pint c = pint_mul(b, b);
  pint d = pint_eq(c, a);
  pint e = pint_mul(d, b);
  pint_measure(e);
}
```

Rather than computing square root directly, our example makes `a`=29929, initializes `b` to all possible 8-bit values with 8-way entanglement (i.e., 2^8 entanglement positions), makes `c` the entangled values for squaring `b`, creates an entangled `d` that is 1 only where `c==a`, and then makes `e` a copy of `b`'s values with all that are not the square root of 29929 zeroed. Of course, no actual FBP operations are triggered until `pint_measure(e)` is used to print the result: `0 173`... and 173 is the square root of 29929.

III. Conclusion

The current work introduces a new model for energy-efficient execution. Efficiency is not based in use of exotic gate-level hardware implementation, but in reducing the number of gate-level operations that must be performed in order to produce each result. This is accomplished through the combination of gate-level compiler optimization technology and novel parallel computer architecture, which could be implemented using conventional gates. Although this work is at a very early stage, it does clearly demonstrate that gate-level complexity of computations can be dramatically reduced – even exponentially – without resorting to exotic quantum phenomena.

A prototype system has been produced, and an example was presented. However, there is much more to be done to optimize and extend both the compiler technology and the architecture. An obvious improvement is that, although the system that runs on a laptop already generates 4,096-bit-wide parallel execution, more parallelism and other adjustments should allow it to be re-targeted to provide appropriate parallelism for multi-core, GPU, and cluster computers. We also plan to expand the set of supported operations, both at the lower levels and by adding support for `pfloat`. In the more distant future, we envision not only a special-purpose architecture tuned for parallel bit pattern computing, but also development of compiler technology for automatic bit pattern parallelization – much like automatic parallelization for more conventional parallel computer execution models.

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] Dietz, H.: How Low Can You Go?. 30th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2017), College Station, Texas, in press with Springer-Verlag Lecture Notes in Computer Science; preprint at `http://aggregate.org/hlcyg.pdf` (October 11, 2017)

[3] Dickinson, A. G. and Denker, J. S.: "Adiabatic dynamic logic," in IEEE Journal of Solid-State Circuits, vol. 30, no. 3, pp. 311-315, March 1995. doi: 10.1109/4.364447

[4] Shende, V. V., Prasad, A. K., Markov, I. L., and Hayes, J. P.: Synthesis of reversible logic circuits, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 22, no. 6, pp. 710-722, June 2003. doi: 10.1109/TCAD.2003.811448

[5] 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).

[6] 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)

[7] 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)

[8] Dietz, H.: A Gate-Level Approach To Compiling For Quantum Computers, 2018 Ninth International Green and Sustainable Computing Conference (IGSC), Pittsburgh, PA, USA, 2018, pp. 1-5. doi: 10.1109/IGCC.2018.8752114

[9] Batcher, K. E.: Design of a Massively Parallel Processor. IEEE Transactions on Computers. C-29 (9): 836–840 (September 1, 1980) doi:10.1109/TC.1980.1675684.

[10] Tucker, L. W., and Robertson, G. G.: Architecture and applications of the Connection Machine. IEEE Computer, Volume 21, Number 8, 26–38 (August 1988)

[11] Harris, M.: Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses (SIGGRAPH '05), John Fujii (Ed.). ACM, New York, NY, USA, Article 50 (2005) DOI: https://doi.org/10.1145/1198555.1198768

[12] Smith, R.S., Curtis, M.J., and Zeng, W.J.: A practical quantum instruction set architecture. arXiv preprint arXiv:1608.03355 (2016)

[13] Cross, A.W., Bishop, L.S., Smolin, J.A., and Gambetta, J.M.: Open quantum assembly language. arXiv preprint arXiv:1707.03429 (July 13, 2017)

[14] Khammassi, N., Guerreschi, G.G., Ashraf, I., Hogaboam, J.W., Almudever, C.G., and Bertels, K.: cqasm v1. 0: Towards a common quantum assembly language. arXiv preprint arXiv:1805.09607 (2018).

[15] Wille, R., Van Meter, R., and Naveh, Y.: IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 1234–1240 (2019)

[16] Svore, K.M., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., and Roetteler, M.: Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. arXiv preprint arXiv:1803.00652 (2018)