Much Ado about Almost Nothing: Compilation for Nanocontrollers

Henry G. Dietz, Shashi D. Arcot, and Sujana Gorantla
Electrical and Computer Engineering Department
University of Kentucky
Lexington, KY 40506-0046
hankd@engr.uky.edu
http://aggregate.org/

Advances in nanotechnology have made it possible to assemble nanostructures into a wide range of micrometer-scale sensors, actuators, and other novel devices... and to place thousands of such devices on a single chip. Most of these devices can benefit from intelligent control, but the control often requires full programmability for each device's controller. This paper presents a combination of programming language, compiler technology, and target architecture that together provide full MIMD-style programmability with per-processor circuit complexity low enough to allow each nanotechnology-based device to be accompanied by its own nanocontroller.

1. Introduction

Although the dominant trend in the computing industry has been to use higher transistor counts to build more complex processors and memory hierarchies, there always have been applications for which a parallel system using processing elements with simpler, smaller, circuits is preferable. SIMD (Single Instruction stream, Multiple Data stream) has been the clear winner in the quest for lower circuit complexity per processing element. Examples of SIMD machines using very simple processing elements include STARAN [Bat74], the Goodyear MPP [Bat80], the NCR GAPP [DaT84], the AMT DAP 510 and 610, the Thinking Machines CM-1 and CM-2 [TMC89], the MasPar MP1 and MP2 [Bla90], and Terasys [Erb].

SIMD processing element circuit complexity is less than for an otherwise comparable MIMD processor because instruction decode, addressing, and sequencing logic does not need to be replicated; the SIMD control unit decodes each instruction and broadcasts the control signals. However, that savings in logic complexity is negligible in comparison to the savings from not needing to replicate the program in the local memory of each MIMD processor. No matter how simple the processor, a long and complex program still requires a very large number of bits of local storage. Program length is somewhat affected by the choice of instruction set architecture, but even the most dense encodings only reduce program size by a small constant factor.

1.1. Meta-State Conversion (MSC)

The ideal would be the simplicity of SIMD hardware with the independent programmability of MIMD. Interpretation of a MIMD instruction set using SIMD hardware is an obvious approach, with a number of non-obvious optimizations required to achieve good efficiency [NiT90][WiH91][DiC93]. One proposal even adds additional hardware broadcast structures to a basic SIMD design to increase efficiency of these techniques [Abu97]. However, any type of interpretation requires a copy of the program local to each processing element, so the size of local program memory brings hardware complexity close to that of a MIMD. In the early 1990s, primarily targeting the MasPar MP-1, we developed basic compiler technology that performs a state-space conversion of a set of MIMD programs into a pure SIMD program with similar relative timing properties: Meta-State Conversion (MSC) [DiK93]. For MSC, it is fairly easy to show that the minimum amount of local memory needed is just enough bits to hold a unique identifier for each processing element, no matter how large the MIMD program may be.

MSC is a state space conversion closely resembling NFA to DFA conversion (as used in constructing lexical analyzers). A MIMD program can be viewed as a state transition diagram in which a particular processor can follow any path it chooses, being in only one state at any given point in time. The MSC algorithm constructs a state transition diagram in which each meta state represents a possible set of original MIMD states that could be held simultaneously by different processors (threads) each executing its own path in the MIMD code. The code within each meta state is guarded (predicated) by testing if the processor was in the original state that would have executed that code. Thus, if one processor might be executing {*A*} while another is executing {*B*}, MSC would represent this by placing both chunks of code into a meta state structured like: {if (*in_A*) {*A*} if (*in_B*) {*B*}}. The result is conversion of the MIMD program into pure SIMD code, although the SIMD code will run slower unless code for different threads can be factored, i.e., the code becomes somethings like: {*AintersectB*; if (*in_A*) {*Aunique*} if (*in_B*) {*Bunique*}}. Thus, factoring code within a meta state is an important step after MSC.

At the time MSC was introduced, we viewed it primarily as a novel way to obtain some additional functionality from a SIMD supercomputer — literally allowing us to program our 16,384 processing element MasPar MP-1 using a shared memory MIMD model while achieving a significant fraction of peak native (distributed memory SIMD) performance. Now, we realize it is actually a fundamentally different model of computation. *Von Neuman Architecture* places code and data in the same memory system; *Harvard Architecture* places code and data in separate but similar memory systems; what one might now call "*Kentucky Architecture*" places data in memory and factors out code, implementing control entirely by selection. The point is that a fully programmable processor can be made *very* small using this model, on the order of 100 transistors: small enough to be used to add fully-programmable intelligence to micrometer-scale devices fabricated using nanotechnology.

1.2. Common Subexpression Induction (CSI)

Although MSC is the enabling compiler technology, efficiency of code resulting from MSC critically depends on the quality of a second technique applied to each block of code: the factoring mentioned above. The Common Subexpression Induction (CSI) [Die92] problem is, given a block of code consisting of *K* separate instruction sequences each with a different selection predicate, minimize the number of instruction broadcast cycles necessary by factoring-out instructions that can be shared by multiple predicates. This is a very difficult problem.

In many ways, the primary contribution of this paper is an operation model better suited to CSI. Clearly, the simplest processing element design will be bit-serial, so applying CSI at the word level, as was originally presented, is likely to miss bit-level opportunities to factor-out operations. There are inherently fewer different types of operations on single-bit values than there are on multi-bit values, so there is an increase in the probability that an operation can be shared by multiple predicates; we can further improve the odds by reducing the number of bit operations available, simultaneously reducing the hardware complexity. We can further simplify the process by making enable be simulated by masking, which makes cost significantly less sequence dependent. The result is that hardware logic minimization techniques can be adapted to perform CSI.

In summary, by adapting the *if-the-else* operation model used in Binary Decision Diagrams (BDDs) [Bry86][Kar88][Gro99] to be both the compilation model and the **only** instruction in the target Instruction Set Architecture (ISA), the analysis, compilation, and target architecture are all made significantly simpler.

The following section describes the target architecture, which is deliberately chosen to be a close match for the operation model used by the new compiler technology. Section 3 describes *BitC*, a simple C dialect designed to facilitate authoring nanocontroller programs. The compilation process is described in section 4, with close attention paid to the CSI algorithm. Preliminary results are summarized in section 5. The conclusion is given in section 6.

2. Nanoprocessor/Nanocontroller Architecture: KITE

We define a *nanoprocessor* to be a Kentucky Architecture processor that has MIMD node circuit complexity several orders of magnitude lower than that of a minimal Von Neuman or Harvard microprocessor. A *nanocontroller* is essentially a nanoprocessor with two additional features:

- The ability to perform local digital and/or analog input/output operations, typically monitoring or controlling another device constructed on the same circuit substrate. For low-temperature nanotechnology fabrication methods, the nanocontroller array might be constructed first and then the nanotechnology devices built on top of their corresponding nanocontrollers.
- Provision for ensuring that real-time timing constraints for specific operations (local input/output) will be met as specified by the nanoprocessor program. Real-time control requires predictable real-time behavior and analog input/output generally would be done using programmed timing and an RC circuit because separate analog converter (ADC or DAC) circuits would be too large. Normally, the MSC process preserves only approximate relative timing between processors, not absolute timing of operations.

Rather than discussing general architectural design constraints for nanocontrollers, the current work is focussed on a very simple new architecture suitable for both nanoprocessors and nanocontrollers: *KITE: Kentucky If-Then-Else*.

There are at least two component modules in the KITE architecture, but more can be added to create a hierarchy of clock rates, thus avoiding running each nanoprocessor at a slow global broadcast clock speed. The complete block-level design using 3 levels is summarized in figure 1.

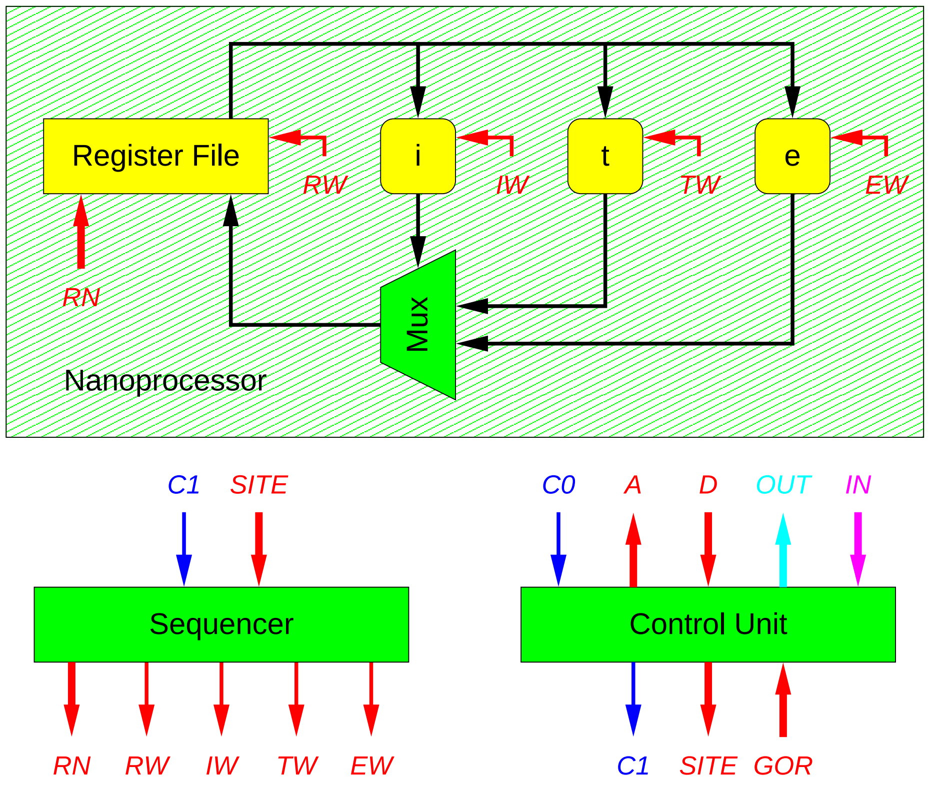

Figure 1: The KITE Architecture

2.1. The Control Unit

In KITE, the *Control Unit* at first appears to be a conventional SIMD control unit. However, it would be more accurate to say that it controls the program memory interface, not the processors.

MSC yields a potentially very large meta-state automaton, which can be viewed as a single sequential program. Each basic block in that program is generally large, and the compiler technology discussed here (see section 4.2) can make basic blocks even larger, if desired. Unlike basic blocks generated for traditional sequential programs, MSC-generated basic blocks typically end in *k*-way branches rather than binary branches; the next meta state is selected by examining the *Global OR (GOR)* of votes from all the processors. Essentially, state transtions in meta-state programs resemble those in code for a VLIW (Very Long Instruction Word) architecture [Ell85].

Given the expectation that meta-state programs will be large, it is appropriate to use off-chip DRAM or PROM, interfaced by conventional address (*A*) and data (*D*) busses. Off-chip memory would be loaded with the meta-state program by a conventional computer host.

A conventional SIMD control unit fetches an instruction at a time, but it is actually more effi cient to fetch a block of code as a single logical entity. For KITE programs, the instruction fetch bandwidth required can be signifi cantly reduced by storing a compressed representation of each basic block in memory. The controller would perform decompression, branch prediction, and instruction cache management treating each basic block as a single unit. This allows the control unit to intelligently prefetch code chunks using a relatively slow clock (*C0*) determined by the external memory system, while internally broadcasting partially decoded instructions (*SITE*s) from cache at a signifi cantly faster rate (*C1*).

2.2. The Sequencers

Although broadcast of decoded signals sounds easy, this has been the clock-speed-limiting factor for most SIMD machines. Even in an on-chip network, huge fanout means being either very slow or very large — relative to nanotechnology fabrication, wires even need to get thick. The purpose of the sequencers is to make a slow broadcast not imply a slow nanoprocessor. Thus, there would be many sequencers, each hosting a moderate number of nanoprocessors.

The *SITE* representation of an instruction actually is a compact form that generates four consecutive clock cycles worth of control information for the nanoprocessors. Thus, the input clock (*C1*) to a sequencer can be as much as four times slower than the nanoprocessor clock. More precisely, a particular sequencer's control line outputs imply a "clock" for the nanoprocessors, but nanoprocessors are only loosely synchronized across sequencers. Put another way, clock skew is managed by higher-level hardware structures working with longer clock cycles that effectively hide the skew of lower levels. Incorporating additional nanoprocessors and sequencers could also provide a means for fault tolerance by disabling the sequencer above each faulty component.

2.3. The Nanoprocessors/Nanocontrollers

The nanoprocessor itself is exceedingly simple: it consists of 1-bit registers, a single register-number decoder, and a 2-to-1 multiplexor. The operation of a multiplexor can be described by analogy to the software concept of an if-then-else; if the value in **`i`** is true, return **`t`**, else return **`e`**. The value returned by the multiplexor can be stored in any register selected.

The *SITE* representation of an instruction is literally four register numbers: the register to store into, the one to load **`i`** with, the one to load **`t`** with, and the one to load **`e`** with. The sequencer simply converts that into a four-cycle sequence using *RN* to specify the register number for the decoder and using the other lines to latch a value into the corresponding register.

For reasons which will become obvious in section 4, “registers” 0 and 1 are not registers, but respectively generate the read-only constants 0 and 1. Similarly, for each application, a KITE nanoprocessor will require specific network connections and local input and output registers; these are addessed like registers starting with register number 2. A minor but important detail is that all network and local input or output registers should be controlled by the values stored, not by the act of decoding their address. (The address decode trick is used in many microprocessor systems, but our nanoprocessors lack true hardware enable/disable logic, so storing the same value that was already present in a register must have no effect.) The minimum number of bits in a KITE register file is thus the sum of 2 constant registers, the number of additional registers needed for network and local input and output, the ceiling of $\log_2$ of the total number of nanoprocessors in the system (for the control state), and the maximum number of ordinary data bits required in any nanoprocessor.

Given the above, a slightly smarter sequencer could be used to opportunistically reduce the total number of clock cycles required from 4 per *SITE* to as few as one — the result store cycle. For example, if the same register number is used to load both **`i`** and **`t`**, the loading of both can be accomplished in a single clock cycle. Further, if the current *SITE* duplicates fields from the previous one, and those fields do not correspond to network or local input or output accesses, the sequencer can skip loading of any of **`i`**, **`t`**, or **`e`**. Such a sequencer would need to buffer incoming *SITE*s to compensate for variability in the rate at which it processes *SITE*s, but execution time would still be predictable because the optimization opportunities depend only on the *SITE* sequence coming from the control unit.

3. Programming Language: BitC

The programming language that we have created for nanocontrollers like KITE is a very small dialect of C that we call *BitC*. It is essentially a sequential programming language, but also supports barrier synchronization, real-time operations, and inter-processor communication.

Like SWARC [DiF99], BitC extends the notion of C types to allow explicit declaration and/or type casting of arbitrary bit precisions using the bitfield syntax from C. For example, **`int:3 a;`** declares **`a`** to be a signed integer value that can be stored in 3 bits. The type system is generally consistent with that of C, although precision of intermediate results is carefully tracked; for example, logical and comparison operators like **`==`** always generate a 1-bit unsigned integer result rather than a “full precision” integer value of 0 or 1. C operators are supported with the standard precedences; a few additional operators like minimum, maximum, and population count also are provided.

BitC supports the usual conditional and looping constructs. However, due to the severely limited amount of data memory associated with each nanoprocessor, stack manipulation for function calls is not supported in the current implementation.

Input and output are accomplished using application-specific reserved registers, which are accessed using variable names assigned by a user-specified mapping. Normally, reservation of special registers would be done before allocation of any ordinary variables. For example, **`int:1 adc@5;`** defines **`adc`** to be allocated starting at bit register 5, simultaneously reserving register 5 so that ordinary variables and temporaries will not be allocated there. Interprocessor communications also are implemented using reserved registers.

4. Compilation

The compilation of BitC for KITE is a complicated process involving a number of transformations. The first step in compiling BitC code is the transformation of word-level operations into simple operations producing single-bit results. These bit-slice operations are optimized and simplified using a variety of techniques from both conventional compiler optimization and hardware logic minimization. The optimized bit-slice versions of all the programs are then logically merged into a single SPMD (Single Program, Multiple Data) program which is Meta-State Converted into guarded SIMD code. Common Subexpression Induction (CSI) is the next step; as discussed in section 4.3, the use of ITEs dramatically simplifies the CSI algorithm. After CSI, the process is essentially complete, but not if too many registers are needed. Thus, the final step is to order the instructions to ensure that max live does not exceed the number of registers available, and then to allocate registers on that basis. The internal form of the instructions after register allocation is identical to the instruction format used by KITE's sequencers.

4.1. Transformation of Word-Level Operations into ITEs

The transformation of word-level operations to bit-level operations is conceptually simple enough, but tedious. The BitC compiler's front end manipulates data at the word level using a data structure for each word value that contains the basic object type (e.g., signed or unsigned), the number of valid bits in the word, and a vector of pointers to bit-level operations or storage cells that yield each bit of the value. Each word-level operation is converted to bit-slice form by invoking a routine that takes word value descriptions as inputs, generates bit-level operations internally, and then returns a description of the results in the form of a word value structure.

Consider transforming the expression **`c=a+b;`** where **`a`**, **`b`** and **`c`** have been declared as **`unsigned int:2 a, b, c;`**. Notationally, let $\mathbf{x}_y$ be the *y*th bit position in the word value **`x`**. Thus, our result value, **`c`** can be written as the word level function:

$$\{\mathbf{c}_1, \mathbf{c}_0\} \leftarrow \{\mathbf{a}_1, \mathbf{a}_0\} + \{\mathbf{b}_1, \mathbf{b}_0\}$$

At the bit level, this turns into two separate functions, one to generate each of the bits of **`c`**. Using the same logic expressions one would find in a typical textbook discussion of constructing a 2's-complement adder circuit, the bit-level representation would be something like:

$$\texttt{c}_0 \leftarrow \texttt{(a}_0\texttt{ XOR b}_0\texttt{)}$$
$$\texttt{c}_1 \leftarrow \texttt{((a}_1\texttt{ XOR b}_1\texttt{) XOR (a}_0\texttt{ AND b}_0\texttt{))}$$

However, this is not a very desirable bit-level format for our purposes. The problem is that even this very short example uses two different operators: XOR and AND. It is not difficult to construct hardware capable of performing any of the 16 possible logic functions of two inputs; a 4-bit lookup table and a 1-of-4 multiplexor suffice. However, having different opcodes complicates both CSI and the logic minimization process. A potentially simpler alternative, commonly used in hardware logic minimization, is to express all functions as if-then-else selections, i.e., as 1-of-2 multiplexors.

Throughout the rest of this paper and the entire BitC and KITE system, an if-then-else tuple is referred to as an *ITE* and, for convenience, we will use C's trinary operator syntax to show the contents of an ITE. Table 1 summarizes equivalents for several familiar logic operations.

Logic Operation	**Equivalent ITE Structure**
`(x AND y)`	`(x ? y : 0)`
`(x OR y)`	`(x ? 1 : y)`
`(NOT x)`	`(x ? 0 : 1)`
`(x XOR y)`	`(x ? (y ? 0 : 1) : y)`
`((NOT x) ? y : z)`	`(x ? z : y)`

Table 1: ITE Equivalents for Familiar Logic Operations

In the BitC compiler, the ITE table data structure is used not only to represent ITE operations, but also nanoprocessor registers. In effect, for a KITE target with *k*-bits of register file, the BitC compiler uses first *k* ITE index values to represent the registers. Thus, ITE index 0 represents the constant 0 and ITE index 1 represents the constant 1. ITEs starting at index 2 represent network registers connecting to other nanoprocessors, local input and output device registers, and bits of user-defined variables. ITEs representing operations start at index *k*; although fewer than *k* registers may have been allocated at this stage, register allocation will make use of these unallocated registers to hold temporary values. In fact, the BitC compiler can be used to determine the minimum usable value for *k* for a particular application, thus serving as a design tool for customizing a KITE hardware implementation.

Rewriting our two-bit addition example using ITEs:

$$\texttt{c}_0 \leftarrow \texttt{(a}_0\texttt{?(b}_0\texttt{?0:1):b}_0\texttt{)}$$
$$\texttt{c}_1 \leftarrow \texttt{((a}_1\texttt{?(b}_1\texttt{?0:1):b}_1\texttt{)?((a}_0\texttt{?b}_0\texttt{:0)?0:1):(a}_0\texttt{?b}_0\texttt{:0))}$$

The multi-valued multi-level logic minimization process using ITEs is very similar to the conventional compiler Common Subexpression Elimination (CSE) process. Just as normalizing the form of tuples for CSE simplifi es the search for matches, the logic minimization process can profi t greatly from normalizing the ITEs. A detailed discussion of normalization of a similar form for logic minimization is given in [Kar88]; as each ITE is generated, we normalize such that the **`i`** component is always a register and registers used in the **`t`** and **`e`** parts have lower numbers. The result of processing the above example is shown graphically in fi gure 2. Registers 0 and 1 hold those constants; registers 2-7 hold **`a`**, **`b`**, and **`c`**.

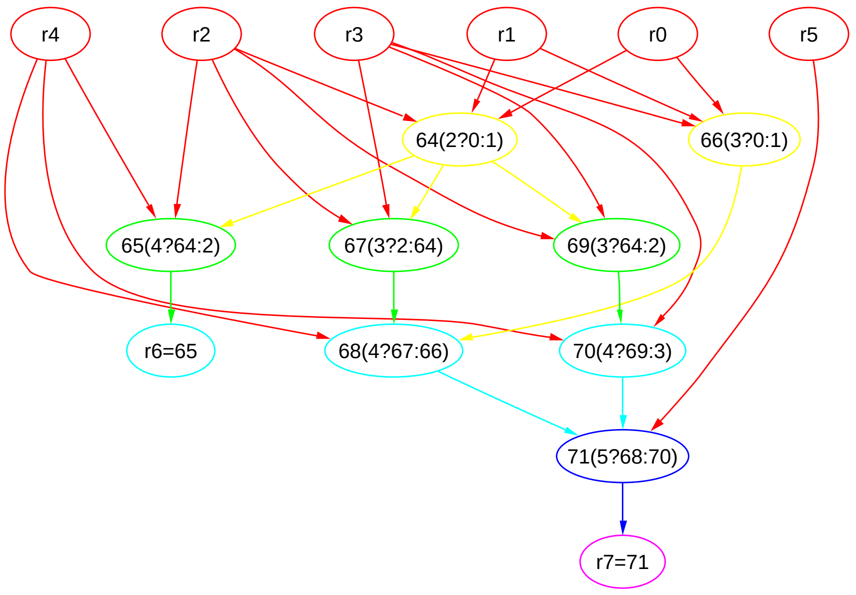

Figure 2: Normalized ITEs for **`unsigned int:2 a,b,c; c=a+b;`**

4.2. Predication using ITEs

Given that an ITE is really the most basic guarded/predicated operation, it is not surprising that conversion of control flow constructs into ITEs is straightforward and effi cient. In [Die00], we discussed how speculative predication could be supported across arbitrary control flow including conditionals, loops, and even recursive function calls. Although speculative execution using KITE would not obtain the same benefi ts as it does targeting IA64, the same techniques can be used to generate larger basic blocks of ITEs that, hopefully, will lead to more effective CSI and a reduction in the total number of control states after the single-nanoprocessor speculative programs have been combined using MSC.

Consider a very simple example:

```
if (a) {b=c; if (d) e=f; else g=h; i=j;} k=l;
```

The nested conditionals can be speculatively executed by nothing more than nested application of `if`-conversion:

```
    b=(a?c:b); e=((a?d:0)?f:e); g=((a?(d?0:1):0)?g:h);
    i=(a?j:i); k=l;
```

This same transformation also is used to implement the guards created by the MSC process, which normally look like single-level `if` sequences testing local state variables. The interesting thing about this transformation is that ITEs created as predication are indistinguishable from ITEs implementing portions of word operations. Thus, the ITE optimization process will naturally minimize both simultaneously.

4.3. Common Subexpression Induction using ITEs

The basic CSI algorithm was first described in [Die92]. That algorithm was implemented and shown to be effective, with reasonable compile times, when targeting the MasPar MP-1. Thus, in describing our new approach using ITEs, it is useful to begin by summarizing the original approach and the assumptions that it makes. There were 8 steps:

Step 1: Construct Guarded DAG. The first step in the CSI algorithm is the construction of a *Guarded DAG* (Directed Acyclic Graph) for the assembly-level operations. The basic DAG structure is typical of those generally used to represent basic blocks in optimizing compilers. The guards are bit masks that indicate to which "thread" each instruction belongs.

Step 2: Inter-thread CSE. Given a guarded DAG, the next step is essentially standard compiler technology for recognizing and factoring-out common subexpressions, but operations with different guards can be factored-out as common subexpressions. Portions of the DAG which, ignoring the guards, would be common subexpressions are in fact equivalent code sequences if the guards say they belong to mutually-exclusive threads. The ITE logic minimization process effectively accomplishes this as a side effect.

Step 3: Earliest and Latest Computation. The CSI search for factoring additional instructions, as proposed in [Die92], is based on permutation of instructions within a linear schedule. A linear schedule was used for two reasons:

1. On machines like the MasPar MP-1, changing the enable status of a processing element is a modal operation requiring execution of at least one instruction. The result is that the precise cost of a potential CSI solution is a function of the number of times the execution sequence changes from one set of threads being enabled to a different set being enabled. For example, an instruction sequence with guards ordered *{ thread0; thread1; thread0; }* is significantly more expensive to execute than *{ thread1; thread0; thread0; }*. Without constructing the order, CSI might combine operations in such a way that execution time is actually increased by factoring!

2. Only instructions that could be physically adjacent in some sequential order could potentially be merged, thus, by searching linear orders it becomes possible to consider only potential pair factorings of instructions adjacent to each other in the schedule under consideration.

A full permutation search on linear schedules is clearly *O(n!)*, feasible only for $n<20$. Instead, the original CSI algorithm used a permutation-in-range search. Each instruction only can appear between its earliest and latest possible schedule slot, the positions of which are computed from the DAG in this step.

Using ITEs, the search need not be based on a linear schedule because reason #1 above does not apply. Because guards are applied directly in each instruction, the cost of a schedule is not directly dependent on the number of times the guard being applied is changed. There only are "second order" effects that may favor one schedule over another, primarily concerning register allocation and opportunistic optimizations made by the sequencers.

Step 4: Classification. The next step in the original CSI algorithm was classification of instructions for potential pairings. This was a fairly complex process involving checking opcodes, immediate operands, ordering constraints (first approximately using earliest and latest and then precisely using th DAG itself), and guard overlap. Classes are used to further prune the search for potential pairings in the linear schedules.

For ITEs, there is only one opcode and no immediate values are embedded in the instruction; there are only two constants, 0 and 1, and they are accessed from registers. Neither is guard overlap a concern, since application of guards is effectively embedded within each instruction.

Step 5: Theoretical Lower Bound. Using the classes and expected execution times for each type of operation, it is possible to compute a good estimate of the lower bound on minimum execution time. This estimate can be used to determine if performing the CSI search is worthwhile — i.e., if the potential for improvement in code execution time by CSI is small, then one might abort the search. The same algorithm is used to evaluate partial schedules to aid in pruning the search.

Step 6: Creation of An Initial Schedule. A viable initial linear schedule is created. In the linear schedule, the N^{th} operation in a schedule is either executed at the same time as the $(N\text{-}1)^{th}$ instruction or in the next "tick."

Step 7: Improving the Initial Linear Schedule. The original CSI technique next applied a heuristic sort to obtain an improved initial schedule.

Step 8: The Search. This is the final, and most complex and time-consuming, step of the original CSI algorithm. It is a heavily pruned permutation-in-range search using pairwise-exchanges of instructions and incrementally updating evaluation of partial schedules.

4.4. The New CSI Algorithm for ITEs

As described above, the guards can be directly absorbed in the computation performed by the ITEs; it then becomes trivial to perform inter-thread CSE. In fact, inter-thread CSE for ITEs *is* ordinary CSE for ITEs, which also can be viewed as a weaker equivalent to the multi-level logic optimization methods published for BDDs. For ITEs, CSI can be accomplished using logic minimization on the ITE DAG: steps #3 through #8 essentially disappear! Thus, the new CSI algorithm for ITEs has only three steps:

Step 1: Generate ITEs. Generate the ITEs, encoding guards for the threads directly, as discussed in section 4.2. ITEs computing the meta-state next state information, as per [DiK93], also are generated.

Step 2: Perform Logic Minimization. Perform essentially standard logic minimization. The goal is simply minimization of the total number of ITEs needed to compute all bit values stored in the block that are live at exit from the block. Thus, this is a multi-valued (each bit stored is a value) multi-level (ITEs can be nested to any depth, as opposed to 2-level AND/OR trees) logic optimization problem. A multitude of logic minimization techniques have been developed since Quine McCluskey surfaced in the 1950s; a good overview of various approaches appears in [HaS96]. Logic optimization is not an easy problem, but there are a variety of efficient algorithms that yield good, if not optimal, results. Our current compiler uses an approach based on the improvements [Kar88] made to [Bry86].

Step 3: Allocate Registers and Schedule Code. Allocate registers for all intermediate values using a linear code schedule that satisfies the constraint that max live never exceeds the number of registers available.

The current BitC compiler has effective implementations of steps 1 and 2, but step 3 requires further study. In many cases, it is easy to find a usable linear schedule and to allocate registers; we are unsure what to do when it is not. One possibility is cracking basic blocks to make smaller DAGs. In summary, the BitC compiler's handling of too-complex basic blocks of ITEs can and should be significantly improved.

5. Results

Although the BitC compiler is still far from generating compressed code blocks for KITE, it is sufficiently complete to allow a wide range of experiments. To facilitate such experiments, the BitC compiler is capable of generating code in a variety of formats, including one suitable for generating graphs using *dot* [GaK02] (figure 2 was created in this way).

From a large number of example codes, here are our preliminary observations:

- Compiler speed is not a problem. Simple test programs are processed in small fractions of a second using a Linux PC.
- Complexity of higher-precision arithmetic operations is a problem. Currently, all the operations appearing within a single basic block are performed by a DAG which is equivalent to a fully combinatorial hardware logic implementation.

Further, the normalized form used is inefficient in representing logic involving **`XOR`** operations, such as those occuring in binary addition and multiplication. Empirically, these operations on values containing more than 12 bits generate too many ITEs. While many nanocontroller applications can be effective using no more than 12-bit precision, a better approach would be for the BitC compiler to introduce additional basic blocks — the equivalent of using multiple clock cycles in a hardware logic implementation.

- Although the BitC compiler does not explicitly perform word-level optimizations, they are very effectively performed as a result of the bit-level analysis. For example, **`int:8 a; a=a+1; a=a-1;`** generates 60 ITEs, but not a single one is left live by the end of the bit-level analysis! Of course, bit-level optimizations within word computations also are recognized.
- The normalized ITE form used in BitC was originally proposed not for logic minimization, but for checking equivalence of expressions: equivalent expressions generate the same ITE DAG. Although the BitC compiler currently does not support non-constant-indexed array references, perhaps this type of equivalence checking could be used for dependence analysis so that arrays could be efficiently supported despite the lack of a hardware indexing method? In the past, SIMD architectures which lacked hardware indexing (e.g., the Thinking Machines CM-2 [TMC89]) generally scanned through *all* array elements to select the one indexed by a non-constant expression.

The precise basic block representation that KITE will use to encode compressed SITE code, multiple exit arcs (multiway branches), and explicit cache management is not yet finalized, so we have not yet simulated whole-system performance. Given the large number of ITEs resulting from arithmetic operations, total processing speed for a KITE system might not be greatly superior to that of a conventional uniprocessor with comparable circuit complexity. However, there is no practical method by which a uniprocessor design could implement the huge number of I/O operations necessary for control of thousands to millions of nanofabricated devices on the same chip.

6. Conclusion

Even using existing micro-scale technology, there are many arrays of devices that could profit from smart control, but for which smart control has been infeasible. It is not feasible to carry thousands of signals to a conventional processor off-chip or even on-chip; nor are conventional processor+memory combinations small enough to be placed with each device. As nanotechnology develops, there will be an increasing need for local intelligent monitoring and control of the new devices. There have been a number of exotic new computational models suggested for using nanotechnology to build computers, but none of these provide a straightforwardly programmable solution to the nanocontroller problem.

It is well known that a SIMD architecture can dramatically lower the circuit complexity per computational node. It also was known that, using meta-state conversion (MSC) [DiK93] and common subexpression induction (CSI) [Die92], SIMD hardware can efficiently execute MIMD programs. The primary difficulty was the complexity of the CSI algorithm and interactions between instruction selection and the effectiveness of CSI. The primary contribution of this paper is the recognition that, by using the bit-level ITE (if-then-else) construct, circuit complexity is reduced and compiler analysis, especially CSI, is dramatically simplified. The ITE representation also facilitates use of existing hardware-oriented logic minimization techniques to simplify the final code.

Preliminary measurements of performance of the BitC compiler clearly demonstrate that low-precision integer control efficiently can be implemented using MSC with an ITE-based target architecture, such as KITE. The circuit complexity of a KITE nanoprocessor is essentially that of a small register file (1-bit SRAM/DRAM or I/O cells plus a possibly shared address decoder), three staging registers (each 1 bit), and a 1-of-2 multiplexor (possibly implemented more like two connected tri-state outputs) with all other resources shared by many nanoprocessors. Thus, given programs that need only a few bytes of local data, complexity can be on the order of 100 transistors per nanocontroller. This design is less suited for general-purpose parallel computing; a somewhat more complex design, with a more powerful function unit, could yield higher performance per unit circuit complexity.

There is much more work to be done to optimize performance of the language, compiler, and details of the KITE architecture and hardware implementation. Much will depend on what control new nanotechnology devices need. This paper marks the start of our research in developing practical nanocontrollers, not the end.

References

[Abu97] Nael B. Abu-Ghazaleh, *Shared Control Multiprocessors - A Paradigm for Supporting Control Parallelism on SIMD-like Architectures*, PhD Dissertation, University of Cincinnati, July 1997.

[Bat74] K. Batcher, "STARAN Parallel Processor System Hardware," *Proc. of the 1974 National Computer Conference*, AFIPS Conference Proceedings, vol. 43, pp. 405-410.

[Bat80] K. Batcher, "Architecture of a Massively Parallel Processor," *Proc. of IEEE/ACM International Conference on Computer Architecture*, 1980, pp. 168-173.

[Bla90] T. Blank, "The MasPar MP-1 Architecture," 35th IEEE Computer Society International Conference (COMPCON), February 1990, pp. 20-24.

[Bry86] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computers, vol. C35, no. 8, pp. 677-691, 1986.

[DaT84] R. Davis and D. Thomas, "Systeolic Array Chip Matches the Pace of High-Speed Processing," reprint from *Electronic Design*, October 31, 1984.

[DiC93] H. G. Dietz and W. E. Cohen, "A Control-Parallel Programming Model Implemented On SIMD Hardware," *Languages and Compilers for Parallel*

Computing, edited by U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Springer-Verlag, New York, New York, pp. 311-325, 1993.

[Die92] H. G. Dietz, "Common Subexpression Induction," Proceedings of the *1992 International Conference on Parallel Processing,* Saint Charles, Illinois, August 1992, vol. II, pp. 174-182.

[Die00] H. G. Dietz, "Speculative Predication Across Arbitrary Interprocedural Control Flow," *Languages and Compilers for Parallel Computing*, edited by L. Carter and J. Ferrante, Springer-Verlag, New York, New York, pp. 432-446, 2000.

[DiF99] H. G. Dietz and R. J. Fisher, "Compiling for SIMD Within A Register," *Languages and Compilers for Parallel Computing*, edited by S. Chatterjee, J,. F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr, and P-C Yew, Springer-Verlag, New York, New York, pp. 290-304, 1999.

[DiK93] H. G. Dietz and G. Krishnamurthy, "Meta-State Conversion," *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 47-56, Saint Charles, Illinois, August 1993.

[Ell85] J. R. Ellis, *Bulldog: A compiler for VLIW Architectures*, ACM Doctoral Dissertation Award, MIT Press, 1985.

[Erb] R. F. Erbacher, *Implementing an Interactive Visualization System on a SIMD Architecture*, University of Massachusetts at Lowell Technical Report, Lowell, MA 01854.

[GaK02] E. Ganser, E. Koutsofi os, and S. North, *Drawing graphs with dot* (dot user's manual), ATT Research, February 4, 2002.

[Gro99] C. Gropl, *Binary Decision Diagrams for Random Boolean Functions*, Ph.D. Dissertation, Humboldt University, Berlin, Germany, May 1999.

[HaS96] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, June 1996.

[Kar88] K. Karplus, *Representing Boolean Functions with If-Then-Else DAGs*, Technical Report UCSC-CRL-88-28, University of California at Santa Cruz, Nov. 1, 1988.

[NiT90] M. Nilsson and H. Tanaka, "MIMD Execution by SIMD Computers," Journal of Information Processing, Information Processing Society of Japan, vol. 13, no. 1, 1990, pp. 58-61.

[TMC89] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, Version 5.1, May 1989.

[WiH91] P.A. Wilsey, D.A. Hensgen, C.E. Slusher, N.B. Abu-Ghazaleh, and D.Y. Hollinden, "Exploiting SIMD Computers for Mutant Program Execution," Technical Report No. TR 133-11-91, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, Ohio, November 1991.