META-STATE CONVERSION†

H. G. Dietz and G. Krishnamurthy
Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

Abstract — *In MIMD (Multiple Instruction stream, Multiple Data stream) execution, each processor has its own state. Although these states are generally considered to be independent entities, it is also possible to view the set of processor states at a particular time as single, aggregate, "Meta State." Once a program has been converted into a single finite automaton based on Meta States, only a single program counter is needed. Hence, it is possible to duplicate the MIMD execution using SIMD (Single Instruction stream, Multiple Data stream) hardware without the overhead of interpretation or even of having each processing element keep a copy of the MIMD code. In this paper, we present an algorithm for Meta-State Conversion (MSC) and explore some properties of the technique.*

1. Introduction

The differences between data parallelism (SIMD execution) and control parallelism (MIMD execution) are at least superficially quite large. In a data parallel program, parallelism is specified in terms of performing the same operation simultaneously on all elements of a data structure; this naturally fits the SIMD execution model. It is also easy to see that, because the abilities of a MIMD are a superset of the abilities of a SIMD, the data parallel model can be extended to MIMD targets [11] [7]. However, the control parallel model suggests that each processor can take its own path independent of all others, and this characteristic seems to require the multiple instruction streams possible only in MIMD execution. Control parallelism is impossible on a SIMD with only one instruction stream... or is it?

There are two basic approaches that might allow SIMD hardware to efficiently support a control parallel programming model: "MIMD emulation" and "meta-state conversion."

1.1. MIMD Emulation

Perhaps the most obvious way to make SIMD hardware mimic MIMD execution is to write a SIMD program that will interpretively execute a MIMD instruction set. In the simplest terms, such an interpreter has a data structure, replicated in each SIMD PE, that corresponds to the internal registers of each MIMD processor. Likewise, each PE's memory holds a copy of the MIMD code to be executed. Hence, the interpreter structure can be as simple as:

Basic MIMD Interpreter Algorithm

1. Each PE fetches an "instruction" into its "instruction register" (IR) and updates its "program counter" (PC).
2. Each PE decodes the "instruction" from its IR.
3. Repeat steps 3a-3c for each "instruction" type:

3.a Disable all PEs where the IR holds an "instruction" of a different type.

3.b Simulate execution of the "instruction" on the enabled PEs.

3.c Enable all PEs.

4. Go to step 1.

The only difficulty in implementing an interpreter with the above structure is that the simulated machine will be very inefficient.

A number of researchers have used a wide range of "tricks" to produce more efficient MIMD interpreters [9], [12], and [3]. However, some overhead cannot be removed:

1. Instructions must be fetched and decoded.
2. Instructions must be accessible to all PEs, hence, each PE typically will have a copy of the entire MIMD program's instructions. In a massively-parallel machine, this wastes a huge amount of memory.
3. There will be some overhead associated with the interpreter itself, e.g., the cost of jumping back to the start of the interpreter loop.

Although problems 1 and 3 merely slow the execution, the second severely restricts the size of MIMD programs. For example, the Purdue University School of Electrical Engineering has a 16K processing element MasPar MP-1 [1] with only 16K bytes of local memory for each PE. Even with very careful encoding, 16K bytes cannot hold a very large MIMD program.

Although meta-state conversion is more difficult to implement and more restrictive in its abilities, it can eliminate even these three overhead problems.

1.2. Meta-State Conversion

In MIMD execution, each processor has its own state. Although these states are generally considered to be independent entities, it is also possible to view the set of processor states at a particular time as single, aggregate, "Meta State." Using static analysis based on the timing described in [6], a compiler can convert the MIMD program into an automaton based on meta states.

Once a program has been converted into the form of a meta-state automaton, it is no longer necessary for each PE to fetch and decode instructions, nor is it necessary that each PE

† This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-91-J-4013 and by the National Science Foundation (NSF) under award number 9015696-CDA.

have a copy of the program in local memory. Only the SIMD control unit needs to have a copy of the meta-state automaton; PEs merely hold data. Further, because there is no interpreter, there is no interpretation overhead. Literally, the meta-state automaton is a SIMD program that preserves the relative timing properties of MIMD execution.

However, just as interpretation has drawbacks, so too does meta-state conversion:

1. If there are N processors each of which can be in any of S states, then it is possible that there may be as many as $S!/(S-N)!$ states in the meta-state automaton. Without some means to ensure that the state space is kept manageable, the technique is not practical.
2. In execution, meta-state transitions are based on examining the aggregate of the MIMD state transitions for all processors.
3. Meta-state transitions are N-way branches keyed by the aggregate of the MIMD state transitions.
4. Dynamic creation of new processes is difficult to accommodate, since construction of the meta-state automaton requires that all possible MIMD states can be predicted at compile time.

Fortunately, we have developed a number of techniques that can control the state space explosion suggested above. Making meta-state transitions based on aggregate information is conceptually simple, but requires some hardware support, e.g., the "global or" of the MasPar MP-1 [1]. The efficient implementation of N-way branches is a difficult problem, but can be accomplished using customized hash functions indexing jump tables [5]. Unfortunately, the fully dynamic creation of processes seems to be impractical — but that is exactly the case in which the interpretation scheme works best. Consequently, this paper focuses on techniques to control the state explosion, and restricts the input MIMD code to be formulated as an SPMD program.

The second section of this paper presents the meta-state conversion algorithm, using an example to clarify the process. Section 3 discusses issues involving how the resulting meta-state automaton can be efficiently encoded for SIMD execution. In section 4, we discuss how the prototype implementation was constructed, and give a simple example of the output generated. Finally, section five summarizes the contributions of this work and directions for future study.

2. Meta-State Conversion

The meta-state conversion algorithm is surprisingly straightforward; perhaps it would be more accurate to say that it is familiar. The process of converting a set of MIMD states that exist at a particular point in time into a single meta state is strikingly similar to the process of converting an NFA into a DFA, as used in constructing lexical analyzers.

To begin, the code for the MIMD processes is converted into a set of control flow graphs in which each node (MIMD state) represents a basic block [2]. Each of these MIMD states has zero, one, or two, exit arcs. A MIMD state with no exit arcs marks the end of that process. A single exit arc represents unconditional sequencing (e.g., an unconditional branch), whereas two exit arcs respectively represent the "TRUE" and "FALSE" successors of that MIMD state (e.g., targets of a conditional branch). In addition, it is assumed that we know in which particular MIMD state each process begins execution; these states are called MIMD start states.

The set of MIMD start states forms the start state of the meta-state automaton. Since each MIMD start state may have up to two successors, each process may pick either of its two possible successors. If we further assume that there may be multiple processes in each MIMD state, it is further possible that *both* successors might be chosen. Hence, for a meta state that consists of one MIMD start state, there may be as many as three meta-state successors. In general, from n MIMD start states, there could be as many as 3^n meta-state successors.

To clarify the operation of the algorithm, we will trace the algorithm's actions on a simple example. The framework for the example is the following SPMD code:

```
if (A) {
    do { B } while (C);
} else {
    do { D } while (E);
}
F
```

Listing 1: Example MIMD (SPMD) Code

It is assumed that all processors begin executing this code simultaneously and that processors computing different values for the parallel expressions A, C, and E are the only sources of asynchrony (i.e., there are no external interrupts).

2.1. Construction of the MIMD Control-Flow Graph

Before meta-state conversion can be applied, the program must be converted into a form that facilitates the analysis. The most convenient form is that of a traditional control-flow graph in which each node represents a maximal basic block. Constructing the control-flow graph in the usual way, code straightening [2] and removal of empty nodes are applied to obtain the simplest possible graph. The result of this is figure 1. State 0 corresponds to block A, state 2 corresponds to B followed by C, state 6 corresponds to D followed by E, and state 9 corresponds to F.

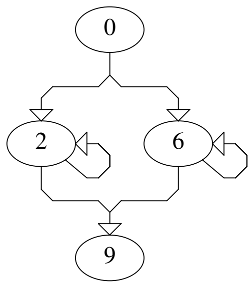

Figure 1: MIMD State Graph for Listing 1

2.2. Handling Of Function Calls

Although our example case does not contain any function calls, it is important that meta-state conversion be applicable to codes that contain arbitrary function calls — perhaps including recursive function invocations. Thus, we need some way to represent function call/return directly using control flow arcs in the MIMD state graph.

In the case of non-recursive function calls, it is sufficient to use the traditional solution of in-line expansion of the function code (i.e., of the MIMD state graph for the function body). Surprisingly, recursive function calls also can be treated using in-line expansion — and an additional "trick" that converts `return` statements into ordinary multiway branches.

Consider the following C-like code fragment in which the `main` program invokes the recursive function `g`:

```
main()
     ...
a:   g();
b:   ...
c:   g();
d:   ...
}

g()
{
     ...
     g();
e:   ...
}
```

Listing 2: Example Recursive Function Call

The only difficulty in in-line expanding `g` is that the target of any `return` statements in `g` is not known until runtime. However, at compile time we can compute the set of all possible `return` targets given that `g` was initially invoked from a particular position.

When in-line expanding the call to `g` from position `a`, we know that any `return` statements within `g` must return to either position `b` or `e`, and can replace the `return` statements with the appropriate multiway branch. Likewise, when in-line expanding `g` called from position `c`, `return` statements are translated into multiway branches targeting `d` or `e`. The result is a call-free control flow graph for the entire program; thus, the meta-state conversion algorithm can ignore the direct handling of function calls without loss of generality.

2.3. Base Conversion Algorithm

The following C-based pseudo code gives the base algorithm for meta-state conversion.

```
meta_state_convert(x)
set x;
{
 /* Given the start meta state x,
    generate the rest of the automaton
 */


 do {
  /* Mark this meta state as done */
  mark_meta_state_done(x);

  /* Add arcs to meta states y| x→y */
  reach(x, x, ∅);

  /* Get another meta state */
  x = get_unmarked_meta_state();

  /* Repeat for that meta state */
 } while (x != ∅);
}

int
reach(start, s, t)
set start, s, t;
{
 /* Make entries for all meta states
    t| start→t
 */
 if (s == ∅) {
  /* All MIMD state transitions from
     within start have been considered,
     hence, t must be a meta state
  */
  make_meta_state_transition(start, t);
 } else {
  /* Select a MIMD state and process
     its transition(s), recursing to
     complete the meta state
  */
  element e, next, fnext;

  e = [e| e ∈ s];
  s = s - {e};
  next = next_MIMD_state(e);
  fnext = next_MIMD_state_if_false(e);

  /* Take each path and both paths */
  if (next) {
   reach(start, s, t ∪ next);
   if (fnext) {
    reach(start, s, t ∪ fnext);
    reach(start, s, t ∪ next ∪ fnext);
   }
  } else {
   reach(start, s, t);
  }
 }
}
```

Applying the above algorithm to our simple example, the resulting meta-state graph is given in figure 2.

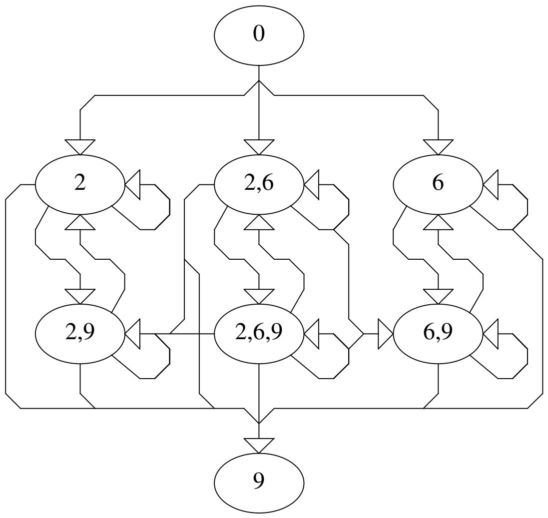

Figure 2: Meta-State Graph for Listing 1

2.4. MIMD State Time Splitting Algorithm

In the base conversion algorithm, we made the assumption that each MIMD state took exactly the same amount of time to execute. However, such an assumption is unrealistic:

- If each instruction is treated as a separate MIMD state, then reasonable size programs will generate unreasonably large automata. This makes the analysis for meta-state conversion much slower and also can result in an impractically large meta-state automaton. In addition, some computers have instruction sets in which even the execution time of different types of instruction varies widely.
- If instead we simply treat each maximal basic block as a MIMD state and ignore the differences in execution time between these blocks, this can result in very poor processor utilization. For example, if a block that takes 5 clock cycles to execute is placed in the same meta-state as one that takes 100 cycles, then the parallel machine may spend up to 95% of its processor cycles simply waiting for the transition to the next meta state.

In other words, the meta-state automaton embodies an *execution time schedule* for the code, and it is necessary that the execution time of each block be taken into account if a good schedule is to be produced.

There are many possible ways in which timing information could be incorporated, but our overriding concern must be keeping the state space manageable, and this greatly restricts the choice. Clearly, the smallest MIMD state automaton results from treating each maximal basic block as a MIMD state; hence, this will be our initial assumption. As the conversion is being performed, we may be fortunate enough to have all the MIMD states merged into each meta state happen to have the same cost. If the costs differ, but do not differ by a significant enough amount, we can ignore the difference.

This leaves only the case of a meta state that contains MIMD states of widely varying cost, for example, the 5 and 100 cycle MIMD states mentioned above. The solution we propose is a simple heuristic that will break the 100 cycle MIMD state into an approximately 5 cycle MIMD state which is unconditionally followed by the remaining portion of the original 100 cycle state. Since this change might also affect the construction of other meta states that had incorporated the original 100 cycle MIMD state, the construction of the meta-state automaton is restarted to ensure that the final meta-state automaton is consistent.

The following pseudocode gives the algorithm for performing MIMD state splitting based on the variation in timing within a meta state. It would be invoked on each meta state as it is created.

```
flag
time_split_state(s)
set s;
{
 /* Determine if time imbalance between
    MIMD states within the meta state s
    is sufficient to time split the more
    expensive MIMD states to get better
    balance; this assumes that each MIMD
    state already has an execution time
    associated with it
 */
 flag didsplit;

 /* Ignore zero time components because
    you can't do anything about them
 */
 s = s - {e| e ∈ s, time(e) == 0};

 /* Get min and max MIMD state times */
 min = min_MIMD_state_time(s);
 max = max_MIMD_state_time(s);

 /* Is enough time wasted to be worth
    splitting?  Not if the difference
    between times is already at noise
    level (split_delta) or if the
    utilization is already sure to be
    greater than an acceptable
    percentage (split_percentage)
 */
 if ((min + split_delta) > max) {
  return(FALSE);
 }
 if (min > ((split_percent*max)/100)) {
  return(FALSE);
 }

 /* Splitting seems useful...  do it */
 didsplit = FALSE;
 while (s != ∅) {
  element e;

  e = [e| e ∈ s];
  s = s - {e};
  if (time(e) > min) {
   /* If possible, split this node into
      two nodes, the first with time
```

```
        ≈ min, the second with the
        remaining time...
     */
     ...
     didsplit = TRUE;
    }
  }

  return(didsplit);
}
```

The splitting of a state is illustrated in the next two figures. The relevant portion of the initial MIMD state graph is:

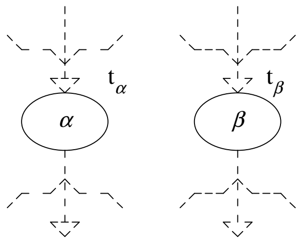

Figure 3: MIMD States Before Time Splitting

Suppose that meta-state conversion would combine states α and β and that β takes much longer to execute than α, i.e., $t_\alpha < t_\beta$. The state splitting algorithm would attempt to convert this portion of the state graph into:

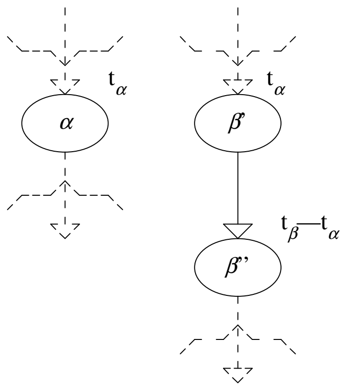

Figure 4: MIMD States After Time Splitting

Thus, states α and β' would be merged — without any idle time being introduced for either thread of execution.

2.5. Meta State Compression Algorithm

Despite the reduction in state space possible using maximal basic blocks and time splitting, the automata created can be very large. Hence, it is useful to find a way to reduce the upper bound on the number of meta states created.

Because MIMD nodes with zero or one exit arc can only increase the state space linearly, the explosion in meta state space is related to the occurrence of MIMD states that have two exit arcs. Each such MIMD state could contribute three meta states: the `TRUE` successor, `FALSE` successor, and both successors. However, if there are many processes in any given MIMD state, it is easy to see that the most probable case is that of both successors. Further, the case of both successors can always emulate either successor, since it has the code for both. Thus, a very dramatic reduction in meta state space can be obtained by simply assuming that both successors are always taken.

```
int
reach(start, s, t)
set start, s, t;
{
 /* Make entries for all meta states
    t| start→t
 */

 if (s == ∅) {
  /* All MIMD state transitions from
     within start have been considered,
     hence, t must be a meta state
  */
  make_meta_state_transition(start, t);
 } else {
  /* Select a MIMD state and process
     its transition(s), recursing to
     complete the meta state
  */
  element e, next, fnext;

  e = [e| e ∈ s];
  s = s - {e};
  next = next_MIMD_state(e);
  fnext = next_MIMD_state_if_false(e);

  /* Always take all possible paths... */
  if (next) {
   if (fnext) {
    reach(start, s, t ∪ next ∪ fnext);
   } else {
    reach(start, s, t ∪ next);
   }
  } else {
   reach(start, s, t);
  }
 }
}
```

Returning to our example code, the meta-state compression algorithm results in a graph with only two meta-states, compared to eight for the uncompressed graph:

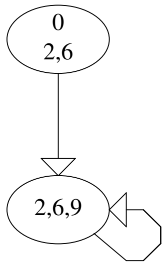

Figure 5: Compressed Meta-State Graph for Listing 1

Notice that meta-state transitions into compressed portions of the graph are unconditional; i.e., there is no need to use a `globalor` to determine what states are present. The disadvantage is that the average meta-state is wider, which implies that the SIMD implementation will be less efficient.

2.6. Barrier Synchronization Algorithm

While the above compression scheme produces very small automata, it does increase overhead somewhat in that each meta state becomes much more complex. Hence, it is useful to seek yet another method to reduce the state space — without adding to the complexity of each meta state. Careful use of barrier synchronization provides such a mechanism.

```
set
barrier_sync(s)
set s;
{
 /* If s is a meta state that contains a
    MIMD state which is a barrier
    synchronization point, then the
    barrier should prevent any
    transitions past that MIMD state.
    Hence, unless all processors have
    reached the barrier (i.e., every MIMD
    state within s is a barrier state),
    simply remove barrier states from s
 */
 set waits;

 /* Construct the set of MIMD barrier
    wait states within s
 */
 waits = {e| e ∈ s, is_barrier_wait(e) == TRUE};

 /* Has everyone reached the barrier? */
 if (waits == s) {
  /* Yes; go into all barrier state */
  return(waits);
 } else {
  /* No; remove barriers from s */
  return(s - waits);
 }
}
```

For example, consider modifying the code framework of listing 1 to contain a barrier sync at the end of the `if`:

```
if (A) {
    do { B } while (C);
} else {
    do { D } while (E);
}
wait;  /* barrier sync. of all threads */
F
```

Listing 3: Listing 1 + Barrier Synchronization

The barrier synchronization does not result in a runtime operation, but rather constrains the asynchrony as defined by the above algorithm. The result is a meta-state graph of the form:

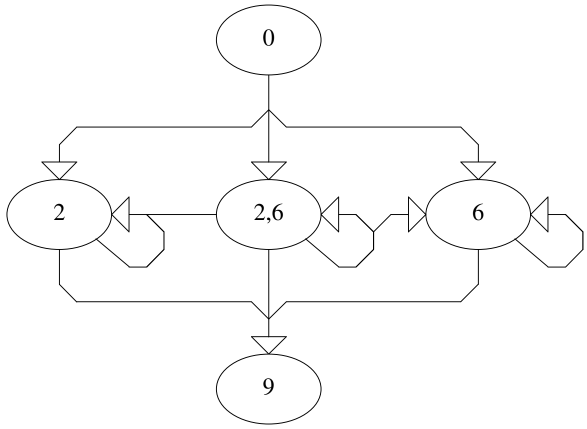

Figure 6: Meta-State Graph for Listing 3

3. SIMD Coding of the Meta-State Automaton

Given a MIMD program that has been converted into a meta-state graph, it is not trivial to find an efficient coding of the meta-state automaton for a SIMD architecture. The meta-state graph does reduce control flow to a single instruction stream, but that instruction stream would appear to execute different types of instructions in parallel — the meta-state graph employs a variation on VLIW semantics.

There are two aspects of the graph that mirror VLIW constructions[1]: the apparently simultaneous execution of different types of instructions and the use of multiway branches generated by merging multiple (binary) branches. Thus, we must efficiently implement these VLIW-like execution structures on SIMD hardware.

3.1. Common Subexpression Induction

Any meta state that merged two or more MIMD states effectively contains multiple instruction sequences that are

[1] The meta-state graph is not suitable for execution on a traditional VLIW because which processing elements execute which instructions is determined statically for VLIW, but dynamically in the graph. I.e., the graph would be appropriate for a VLIW in which each processing element could select at runtime which instruction field it would execute, rather than having each processing element statically associated with a particular instruction field.

supposed to execute simultaneously. Given that it is impossible for a traditional SIMD machine to simultaneously execute different types of instructions on different processing elements, it would appear that these operations will have to be serialized. However, it is quite possible and practical that any operations that would be performed by more than one sequence can be executed in parallel by all processors. Common subexpression induction (CSI) [4] is an optimization technique that identifies these operations and "factors" them out.

The CSI algorithm analyzes a segment of code containing operations executed by any of multiple threads (enabled sets of SIMD PEs). From this analysis, it determines where threads can share the same code and what cost is associated with inducing that sharing. Finally, it generates a code schedule that uses this sharing, where appropriate, to achieve the minimum execution time. Unfortunately, this implies that the CSI algorithm is not simple.

The algorithm can be summarized as follows. First, a guarded DAG is constructed for the input, then this DAG is improved using inter-thread CSE. The improved DAG is then used to compute information for pruning the search: earliest and latest, operation classes, and theoretical lower bound on execution time. Next, this information is used to create a linear schedule (SIMD execution sequence), which is improved using a cheap approximate search and then used as the initial schedule for the permutation-in-range search that is the core of the CSI optimization.

3.2. Multiway Branch Encoding

At the end of each meta-state's execution, a particular type of multiway branch must be executed to move the SIMD machine into the correct next meta state. Before discussing the encoding of these multiway branches, it is useful to specify the precise semantics of meta-state transitions, so that an optimal coding can be achieved. The following defines the possible types of meta-state transitions.

3.2.1. No Exit Arc

A meta state without an exit arc is a terminal node, i.e., it represents the end of the program's execution. Thus, it is implicitly followed by a return to the operating system. There is no difficulty in generating code to implement this.

3.2.2. Single Exit Arc

If there is a single exit arc from a meta state, the code for that meta state is is followed by a `goto` (aka, `jump`) to the code for the target meta state. Again, it is simple to generate an efficient coding.

Notice that all entries to compressed meta states fall into this category.

3.2.3. Multiple Exit Arcs

If there are multiple exit arcs from a meta state, then the aggregate of the "pc" values for each of the processing elements must be used to determine the next state. For example, when, at the end of executing a meta state, some processing elements have "pc" value 2 and others have "pc" value 6, meta state {2,6} is the next state. In order to efficiently collect this aggregate, each possible "pc" value is assigned a bit; thus, a `globalor` of the "pc" values from all processors determines the aggregate.

3.2.4. Multiple Exit Arcs Involving Barriers

The treatment of multiple exit arcs must be slightly adjusted if some, but not all, of the processing elements have reached a barrier at the time a meta state's execution completes. For example, in figure 6 the transitions from meta states 2, {2,6}, and 6 into 2, {2,6}, and 6 would not be sufficient if even one processing element had reached the barrier (i.e., meta state 9). Consequently, the processing elements are allowed to set their "pc" value to 9, but they are not permitted to enter meta state 9 unless all "pc"'s are 9.

This is accomplished by a simple check to see if (`globalor` pc) is contained within the set of all barrier states. If it is, then the state transition proceeds normally. Otherwise, the next meta state is determined by subtracting the set of all barrier states from the result of the `globalor`.

3.2.5. Restricted Dynamic Process Creation

Although the completely static nature of meta-state conversion makes it impossible to efficiently support forking of new processes to execute different programs, a minor encoding trick can be used to implement a restricted form of dynamic process creation. This restricted type of `spawn` instruction looks just like a conditional jump, except the semantics are that both paths must be taken (i.e., the compressed meta state transition rule). One exit is taken by the original processes, the other by the newly created processes.

Initially, processing elements that are not in use would be given a "pc" value indicating that they are not in any meta state. When a `spawn(`x`)` instruction is reached by N processing elements, the original N processing elements do not change their pc values, but N currently-disabled processing elements are selected and their pc values are set to x. No other changes are needed, provided that the number of processes requested does not exceed the number of processors available.

Note further that processors that complete their processes early can be returned to the pool of free processors by simply executing a `halt` instruction to set their pc value to indicate that they are not in any meta state.

3.3. Allocation of Bits for "pc" Values

Although it is easy to implement each "pc" value by assigning a different bit to each MIMD state, this would result in impractically long bit strings for large meta state automata. Thus, although conceptually a different bit is used to represent each MIMD state, bits actually can be reused without changing the basic conversion algorithm. This bit allocation problem is similar to that of allocating registers to values where the number of values can be larger than the number of registers. However, unlike register allocation, there is no concept of "spilling" a bit position; if an allocation is not found using *maxbit* bits, we must increase the number of bits. Fortunately, it is unlikely that *maxbit* will need to be large; in our preliminary experiments, it never was necessary to use more than 8 bits.

Intuitively, there are just two rules that govern the reuse of a bit to represent multiple MIMD states:

1. No two MIMD states contained within the same meta state can be allocated the same bit. If this were violated, it would be impossible to tell which code within that meta state should be executed by each processing element.
2. No two meta states which are successors of the same meta state (i.e., which are sibling meta states) can be allocated the same bit pattern. If two siblings had the same bit pattern, the meta state automaton would not be able to decide which of these sibling meta states to execute.

The algorithm which we have implemented is given below. It applies rule 1 directly, but uses a safe approximation to rule 2. The approximation is simply that no two distinct MIMD states that appear in sibling meta states are allocated the same bit.

```
allocate_bits(maxbit)
int maxbit;
{
 /* Allocate bits to "pc" values, using at
    most maxbit bits.  Returns with ERROR
    if need more than maxbit bits.
 */
 set M, C, S;
 int pos, pat, n, c_n, bit[];

 M = {m| m ∈ meta states};

 /* Assign start state bit 0 */
 m0 = (start meta state ∈ M);
 bit[m0] = 2^0;

 for (m| m ∈ (M - m0)) {
  /* Set of all next meta states of m */
  C = {c| c ∈ M, m→c };
  S = {s| s ∈ C, bits of s have been allocated };
  C = C - S;

  /* OR patterns of all members of S */
  pat = OR(p| p is a pattern of some s ∈ S);
  n = # of distinct MIMD states in S;
  if (# of 1 bits in pat ≠ n) {
   /* Some MIMD states have same bit pattern */
   return(ERROR);
  }

  if (C != ∅)) {
   /* There are some MIMD states whose
      bit patterns need to be allocated
   */
   for (c| c ∈ C) {
    if (c has more than 1 MIMD state) {
     bit[c] = OR(p| p is pattern of MIMD state∈c);
     c_n = # of MIMD states in c;
     if (# of 1 bits in bit[c] ≠ c_n) {
      return(ERROR);
     }
     pat = OR(pat, bit[c]);
    } else {
     /* If only 1 MIMD state, assign pattern */
     if (# of 0 bits in pat == 0) {
      /* No free bit to allocate */
      return(ERROR);
     }
     pos = position of first 0 bit in pat;
     bit[c] = 2^pos;
    }
   }
  }
 }

 return(NO_ERROR);
}
```

4. Implementation

The current prototype meta-state converter does not directly generate executable SIMD code from a MIMD-oriented language. Instead, it simply outputs a set of meta-state definitions. Each of these meta states must then be common subexpression inducted and the meta-state transitions (multiway branches) must be encoded using hash functions. However, these last two steps are implemented by two software tools developed earlier:

- A common subexpression inductor, described in [4].
- A hash function generator, described in [5].

Thus, in this paper we will confine the discussion to the implementation of the prototype meta-state converter. The meta-state converter was written in C using PCCTS [10] and is actually a modified version of the `mimdc` compiler described in [3].

4.1. The Input Language

The language accepted by the meta-state converter is a parallel dialect of C called MIMDC. It supports most of the basic C constructs. Data values can be either `int` or `float`, and variables can be declared as `mono` (shared) or `poly` (private) [11].

There are two kinds of shared memory reference supported. The `mono` variables are replicated in each processor's local memory so that loads execute quickly, but stores involve a broadcast to update all copies. It is also possible to directly access `poly` values from other processors using "parallel subscripting":

```
x[||i] = y[||j] + z;
```

would use the values of `i`, `j`, and `z` on this processor to fetch the value of `y` from processor `j`, add `z`, and store the result into the `x` on processor `i`. In addition to allowing use of shared memory for synchronization, MIMDC supports barrier synchronization [6] using a `wait` statement.

4.2. The Conversion Process

A brief outline of the prototype implementation is:

1. As the PCCTS-generated parser reads the source code, a traditional control-flow graph whose nodes are expression trees is built. This control-flow graph is constructed in a "normalized" form that ensures, for example, that loops are all of the type that execute the body one or more times, rather than zero or more (e.g., by replicating some code and inserting an additional `if` statement).

2. The control-flow graph is straightened and empty nodes are removed. This maximizes the size of the nodes.
3. The meta-state conversion algorithm is applied. Except for the handling of function calls, the prototype implements the full algorithm.
4. The resulting meta-state graph is straightened and output.

The current prototype implementation does not perform the final encoding of the meta-state automaton. Hence, a CSI tool [4] and a tool for finding hash functions [5] are applied by hand to produce the final SIMD code in MPL.

4.3. An Example

To illustrate how the prototype meta-state converter works, consider the MIMDC program presented in listing 4. This example has the same control structure given in listing 1, but is a complete program, so that the actual code generated can be given.

```
main()
{
    poly int x;

    if (x) {
        do { x = 1; } while (x);
    } else {
        do { x = 2; } while (x);
    }

    return(x);
}
```

Listing 4: Example MIMDC Program

Without compression or time cracking, the resulting meta-state SIMD automaton, written in MPL [8] for the MasPar MP-1 [1], is given in listing 5 (note that the algorithm in section 3.3 was not applied). The code within each meta state is simple SIMD stack code using MPL macros for each operation. The only surprising stack operation is `JumpF(x, y)`, which simply sets each processing element's `pc` equal to 2^x if the top-of-stack value is "FALSE" or to 2^y if it is "TRUE." The `apc` is simply the aggregate obtained by oring the values of all the individual `pc`s; the switch at the end of each meta state simply employs a customized hash function to ensure that the multiway branch is implemented efficiently. For example, at the end of meta state 0 (i.e., `ms_0`), instead of a `switch` on `apc` with cases for `BIT(2)|BIT(6)`, `BIT(6)`, and `BIT(2)`, a hash function is applied to make the case values contiguous so that the MPL compiler will use a jump table to implement the `switch`.

5. Conclusions

Although meta-state conversion is a complex and slow process, it does provide a mechanical way to transform control-parallel (MIMD) programs into pure SIMD code. Further, the execution of the meta-state program can be very efficient. In particular, fine-grain MIMD code is generally inefficient on most MIMD machines due to the cost of runtime synchronization, but synchronization is implicit in the meta-state converted SIMD code, and hence has no runtime cost.

While the prototype implementation demonstrates the feasibility and correctness of the meta-state conversion algorithm, it does not yet automate the process of generating the final SIMD code. Future work will integrate the code generation process and will benchmark performance on "real" programs.

References

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

[2] J. Cocke and J.T. Schwartz, *Programming Languages and Their Compilers,* Courant Institute of Mathematical Sciences, New York University, April 1970.

[3] H.G. Dietz and W.E. Cohen, "A Control-Parallel Programming Model Implemented On SIMD Hardware," in Proceedings of the *Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[4] 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.

[5] H.G. Dietz, "Coding Multiway Branches Using Customized Hash Functions," Technical Report TR-EE 92-31, School of Electrical Engineering, Purdue University, July 1992.

[6] H.G. Dietz, M.T. O'Keefe, and A. Zaafrani, "An Introduction to Static Scheduling for MIMD Architectures," Advances in Languages and Compilers for Parallel Processing, edited by A. Nicolau, D. Gelernter, T. Gross, and D. Padua, The MIT Press, Cambridge, Massachusetts, 1991, pp. 425-444.

[7] M. S. Littman and C. D. Metcalf, *An Exploration of Asynchronous Data-Parallelism*, Technical Report, Yale University, July 1990.

[8] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2*, Document Number 9302-0001, Sunnyvale, California, November 1991.

[9] 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.

[10] T.J. Parr, H.G. Dietz, and W.E. Cohen, "PCCTS Reference Manual (version 1.00)," *ACM SIGPLAN Notices*, Feb. 1992, pp. 88-165.

[11] M.J. Phillip, "Unification of Synchronous and Asynchronous Models for Parallel Programming Languages" Master's Thesis, School of Electrical Engineering, Purdue University, West Lafayette, Indiana, June 1989.

[12] 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.

```
ms_0:
  if (pc & BIT(0)) {
    Push(0) LdL JumpF(6,2)
  }
  apc = globalor(pc);
  switch (((-apc) >> 5) & 3) {
  case 1: goto ms_2_6;
  case 2: goto ms_6;
  case 3: goto ms_2;
  }

ms_2:
  if (pc & BIT(2)) {
    Push(1) Push(0) LdL Push(12) StL
    Pop(2) Push(4) LdL JumpF(9,2)
  }
  apc = globalor(pc);
  switch (((-apc) >> 8) & 3) {
  case 1: goto ms_2_9;
  case 2: goto ms_9;
  case 3: goto ms_2;
  }

ms_9:
  if (pc & BIT(9)) {
    Push(4) LdL Ret(3)
  }
  /* no next meta state */
  exit(0);

ms_2_9:
  if (pc & BIT(2)) {
    Push(1) Push(0) LdL
    Push(12) StL Pop(2)
  }
  if (pc & (BIT(2) | BIT(9))) {
    Push(4) LdL
  }
  if (pc & BIT(2)) JumpF(9,2)
  if (pc & BIT(9)) Ret(3)
  apc = globalor(pc);
  switch (((-apc) >> 8) & 3) {
  case 1: goto ms_2_9;
  case 2: goto ms_9;
  case 3: goto ms_2;
  }

ms_6:
  if (pc & BIT(6)) {
    Push(2) Push(0) LdL Push(12) StL
    Pop(2) Push(4) LdL JumpF(9,6)
  }
  apc = globalor(pc);
  switch (((-apc) >> 8) & 3) {
  case 1: goto ms_6_9;
  case 2: goto ms_9;
  case 3: goto ms_6;
  }

ms_6_9:
  if (pc & BIT(6)) {
    Push(2) Push(0) LdL
    Push(12) StL Pop(2)
  }
  if (pc & (BIT(6) | BIT(9))) {
    Push(4) LdL
  }
  if (pc & BIT(6)) JumpF(9,6)
  if (pc & BIT(9)) Ret(3)
  apc = globalor(pc);
  switch (((-apc) >> 8) & 3) {
  case 1: goto ms_6_9;
  case 2: goto ms_9;
  case 3: goto ms_6;
  }

ms_2_6:
  if (pc & BIT(2)) Push(1)
  if (pc & BIT(6)) Push(2)
  if (pc & (BIT(2) | BIT(6))) {
    Push(0) LdL Push(12) StL
    Pop(2) Push(4) LdL
  }
  if (pc & BIT(2)) JumpF(9,2)
  if (pc & BIT(6)) JumpF(9,6)
  apc = globalor(pc);
  switch (((apc >> 6) ^ apc) & 15) {
  case 5: goto ms_2_6;
  case 8: goto ms_9;
  case 9: goto ms_6_9;
  case 12: goto ms_2_9;
  case 13: goto ms_2_6_9;
  }

ms_2_6_9:
  if (pc & BIT(2)) Push(1)
  if (pc & BIT(6)) Push(2)
  if (pc & (BIT(2) | BIT(6))) {
    Push(0) LdL Push(12) StL Pop(2)
  }
  if (pc & (BIT(2) | BIT(6) | BIT(9))) {
    Push(4) LdL
  }
  if (pc & BIT(2)) JumpF(9,2)
  if (pc & BIT(6)) JumpF(9,6)
  if (pc & BIT(9)) Ret(3)
  apc = globalor(pc);
  switch (((apc >> 6) ^ apc) & 15) {
  case 5: goto ms_2_6;
  case 8: goto ms_9;
  case 9: goto ms_6_9;
  case 12: goto ms_2_9;
  case 13: goto ms_2_6_9;
  }
```

Listing 5: Meta-State Converted Example

Table of Contents