Optimizing the Emulation of MIMD Behavior on SIMD Machines

Peter Sanders
Universität Karlsruhe, D-76128 Karlsruhe
E-mail: `sanders@ira.uka.de`

Abstract

SIMD computers have proved to be a useful and cost effective approach to massively parallel computation. On the other hand, there are algorithms which are very inefficient when directly translated into a data-parallel program. This paper presents a number of simple transformations which are able to reduce this *SIMD overhead* to a moderate constant factor. In particular, this factor is often much smaller than the factor incurred by the previously used technique of interpreting machine instruction sets. The paper also introduces techniques for reducing the remaining overhead using Markov chain models of control flow. The optimization problems involved turn out to be **NP**-hard in general but there are many useful heuristics, and closed form optimizations for a probabilistic variant.

1 Introduction

Single Instruction Multiple Data computers are a quite effective approach to massively parallel computation. Since instructions are stored and decoded centrally, the processing elements (PEs) do not need instruction memory or a control unit. This makes it feasible to integrate many simple PEs on an area normally required for one single processor of a MIMD machine. In addition, the synchronous nature of SIMD processing often makes programming easier.

On the other hand, there are algorithms which are very inefficient when directly translated into a data-parallel program. This happens whenever there is a loop with a small number of iterations on most PEs for which there is at least one PE with a much larger number of iterations. In this case, all PEs have to wait until the last PE finishes. For programs whose execution time is dominated by such loops, a SIMD computer performs very poorly. A simple example is the task to compute the Mandelbrot set [13]; the number of iterations necessary to compute a point varies widely. In [6] methods for restructuring nested loops which are typical for numerical applications are discussed. Even more complex control flow patterns can be observed for irregular non-numeric applications. In the following, depth-first tree search is used as an example.

The pseudo-code in Figure 1 gives the kernel of a non-recursive, parallel depth-first search algorithm which searches for leaves constituting a solution. It is assumed that a load balancer takes care that each PE gets a different subtree of the search space (e.g. refer to [11]). A subtree is represented by its root. Interior nodes are expanded by pushing their first successor on the stack. For leaf nodes,

```
Let s be a stack containing only the root node
of a subtree to be processed on this processor
LOOP
    IF isLeaf(top(s)) THEN
        IF isSolution(top(s)) THEN printSolution(top(s))
        WHILE noMoreSiblings(top(s)) DO
            pop(s)
            IF isEmpty(s) THEN stop
        replace top(s) with nextSibling(top(s))
    ELSE push(firstSuccessor(top(s)))
```

Figure 1: Nonrecursive generic depth-first search.

it is checked whether they constitute a solution. Then the program backtracks to the next node with unsearched siblings.

For many problems, the while-loop responsible for backtracking performs very few iterations on the average; but on some PEs the number of iterations may be the full depth of the tree. Additional complications could be introduced by heuristics, or by loops inside the application-specific functions isLeaf, firstSuccessor, etc.

This paper presents techniques for removing the offending loops and producing an equivalent, more efficient program. It is a generalization of [10] which focuses on a probabilistic model of SIMD-emulation for which closed-form solutions can be derived. Here we show that even a special case of the more accurate deterministic emulation models leads to **NP**-hard optimization problems. We also give a more complete treatment of heuristic techniques which can be used to improve performance manually or by the compiler.

Section 2 presents the basic techniques and a number of ideas for optimization. Then Section 3 models control flow of programs using Markov chains. The results make it possible to find optimizations with less trial and error. Section 4 discusses a different approach to transforming programs into synchronous form which is used by many other researchers. It turns out that it can be viewed as a special case of our transformation approach and that the Markov chain models again yields interesting insights. It is also proved that even for this special case, one of the basic optimization problems considered is **NP**-hard. Section 5 summarizes some results on probabilistic test loops for which the corresponding optimization problem has simple closed form solutions. In Section 6 we report on implementation experiences where these techniques are applied to open problems in cellular automata theory and to game tree search. Section 7 summarizes the results.

2 Transformation into synchronous form

The general idea for "SIMD-izing" an algorithm is very simple. Every MIMD program can be transformed into a SIMD program of the general form of a *test loop* given in Figure 2. The o_i are *elementary operations* (still to be determined) which do not contain loops. (Loops with a globally known number of iterations are no problem.) The statement **IF** g_i **THEN** o_i is a *test* for operation o_i.[1]

[1]More generally, for many applications it makes sense to decompose the program (possibly dynamically) into a sequence of test loops — each with a different arrangement of tests — but,

```
initialization
LOOP
    IF g1 THEN o1
    IF g2 THEN o2
    ⋮
    IF gn THEN on
```

Figure 2: Test loop suitable for a SIMD machine.

```
initialize as in Algorithm 1
state := Search
LOOP
    IF state = Search THEN
        IF isLeaf(top(s)) THEN
            IF isSolution(top(s)) THEN state := Solution
            ELSE state := GetNextChoice
        ELSE state := MakeChoicePoint
    IF state = MakeChoicePoint THEN
        push(firstSuccessor(top(s))); state := Search
    IF state = GetNextChoice THEN
        IF noMoreSiblings(top(s)) THEN
            pop(s)
            IF isEmpty(s) THEN stop
            state := GetNextChoice
        ELSE top(s) := nextSibling(top(s)); state := Search
    IF state = Solution THEN
        printSolution(top(s)); state := GetNextChoice
```

Figure 3: Depth-first search controlled by an automaton.

If the control logic of an algorithm can be implemented by a finite automaton then the test loop can be constructed by introducing one elementary operation for each state. For example, Algorithm 1 can be transformed into the test loop depicted in Figure 3. The key observation is that every problem can be cast into this shape:

1. Without loss of generality assume that all PEs run the same process (Single Program Multiple Data programming model).

2. Eliminate calls to procedures which contain loops with a varying number of iterations. This can be done by inlining or by replacing procedure calls by appropriate stack manipulations and control structures.

3. Implement loop control by goto-statements.

4. The code sections between goto-labels now constitute the set of elementary operations, i.e., a code section label: code is replaced with **IF** state = LABEL **THEN** code and a jump goto label is replaced with the assignment state := LABEL. (The labels are replaced with unique constants.)

since the loops can be investigated one at a time, we can restrict ourselves to one loop.

This transformation could, for example, be performed by a compiler. For manual use however, it is better to step back and select states which have a meaningful interpretation in the application domain.

Note that there are two different kinds of control flow. One is the control flow of the problem to be emulated. We call this the asynchronous control flow or simply control flow. The other is the control flow of the test loop which deterministically cycles through the tests. Here we talk about the position in the test loop.

A test loop along the pattern of Figure 2 still contains two sources of inefficiency: First, the required number of iterations through the outer loop can vary from PE to PE. But this problem of *load imbalance* is a general problem of parallel computing which also occurs on MIMD computers. Therefore, load balancing strategies are not discussed here. Furthermore, during a test for an operation o_i, all PEs for which g_i does not hold, are deactivated. (We call this an *unproductive test.*) This remaining *SIMD overhead* depends on the complexity of the program and not on the problem size. Therefore, every MIMD program can be emulated by a SIMD program with constant overhead. Still, in practice it is important to keep this constant small in order to be competitive with MIMD machines.

2.1 Optimizing the test loop

So far, we have always considered test loops which test for every operation exactly once in some arbitrary order. But we are free to select any order of tests. We can even duplicate tests if this helps. As a general heuristics, it is a good idea to test for cheap, frequently needed operations more often than for expensive, rarely needed ones. Also, the tests should be ordered in such a way that a maximum total number of productive tests per iteration of the test loop is performed.

For example, let us assume that interior nodes of the trees to be traversed by Algorithm 3 have many descendents, that there are very few solutions and that operation Solution takes 10 units (of time) while all other operations cost 1 unit. The control flow is therefore dominated by subsequences of the form Search; GetNextChoice; Search; GetNextChoice; ... The test loop Search; GetNextChoice; MakeChoicepoint; Solution takes 13 units and about 2 productive tests per iteration are performed. The test loop Search; GetNextChoice; Search; GetNextChoice; MakeChoicepoint; Solution on the other hand, takes 15 units but about 4 productive tests per iteration are performed — it is almost two times more efficient.

Similar ideas are discussed in [2, 1, 8]. In [8] it is argued that duplicating tests is useless since some PEs are actually delayed due to large deviations from the average control flow. But this is not always a problem. Often all PEs have quite similar control flow characteristics, and even if there are PEs which are delayed, this only means that operation duplication increases *load imbalance* which only results in a longer *execution time* if the load balancer is not able to cope with the additional imbalance. Depth-first tree search for example, can be a very irregular problem anyway and a load balancer which works for a simple test loop has no trouble handling the minor additional imbalance from test duplication.

2.2 Selecting Operations

So far, we have assumed that the set of operations is fixed. However, there are a number of useful transformations on operations which help to increase efficiency:

Splitting: An operation of the form

α
IF c **THEN** β
γ
can be replaced with the following three operations:

o_α: α; $\mathsf{state} :=$ (**IF** c **THEN** o_β **ELSE** o_γ)
o_β: β; $\mathsf{state} := o_\gamma$
o_γ: γ

This is useful if the branch β is rarely taken or very expensive. Since β is an operation of its own now, it can be tested for less frequently than other cheaper or more important operations. For the case discussed in Section 2.1 for example, it is a good thing to have an independent operation Solution instead of making it part of the operation Search. Sometimes the inverse operation of incorporating an operation into another is also useful in order to decrease control overhead. It should have become clear now that it is not clear at all when to apply what transformation. This is the reason why Sections 3 and 4 develop mathematical models which help to make these decisions.

Simplification: In traditional programs, most code need not be fine-tuned since only the small fraction of code in the inner loop is critical. In our approach the *entire* test loop is the inner loop. So, tuning rarely used operations can have an unexpected impact on performance. On the other hand, traditional programs often profits from optimized treatment of some special cases. In a SIMD program however, this approach may backfire since the code for the special case incurs additional SIMD overhead. In a sense, *removing* optimizations is sometimes the better optimization.

Merging: If two operations are almost identical like

o_1: α; $\mathsf{state} := o'_1$
o_2: α; $\mathsf{state} := o'_2$

they can be merged into the single operation

o_{12}: α; $\mathsf{state} := \mathsf{follow}$

if other operations assign the proper value to follow before setting state to o_{12}. This transformation reduces the number of operations and therefore decreases SIMD overhead. Often, splitting and simplification of operations can be used to produce candidates for merging. Essentially, merging is a primitive kind of procedure call and the idea can be expanded to nested calls and recursion by introducing a return stack. In [2] a method called *common subexpression induction* is mentioned which automatically recognizes mergeable parts of code.

3 Modeling control flow with Markov chains

It can involve a lot of trial and error to apply the optimizations in Sections 2.1 and 2.2. Therefore, this section develops mathematical tools which help to select appropriate transformations.

The first step is to abstract from the problem of load balancing which is application dependent and not a specific problem of SIMD computing. This can conveniently be done by assuming infinite load on every processor. Performance is then naturally expressed as the average number of productive tests per unit of time (throughput). The choice of the next operation depends on the current operation and some unknown (hidden) computation we assume to be random. Under these assumption the operations can be identified with the states of a Markov chain. Let p_{ij} designate the transition probability i.e. the probability that the operation o_i follows o_j in the asynchronous control flow. Let c_i be the cost of testing for operation o_i. This model was developed independently from [8] where it is used in a slightly different and simplified setting.

A quite different model is considered in [3]. Here the instruction trace of all PEs over the entire computation is supposed to be known. A very long test loop specifically tuned for this specific instruction trace is to be determined. We have cooperated with the group of Professor Wilsey in proving that this problem is **NP**-hard but still prefer the simpler model which only uses measurements of operation frequencies. This makes realistically short test loops possible which work for a range of related problem instances rather than one specific computation for which a trace was measured.

3.1 Assessing the performance of a test loop

Using the Markov chain model above it is possible to predict the performance of a candidate test loop. This can be done using a kind of symbolic execution: Given the transition probabilities and a vector containing the probabilities that the asynchronous control flow is currently in a given state, it is possible to compute the impact of the next test on this vector. By keeping track of the cost of tests and the fraction of PEs which do productive tests and by iterating a few times through the test loop, a cost function expressing the average cost per productive test can be approximated. (For details refer to [10].)

This is equivalent to modeling asynchronous control flow *and* a specific test loop by a Markov chain: A state s_{ik} represents a situation where o_i is the operation needed next and k is the current position in the test loop. Using this approach, the cost function can be computed by solving a large, sparse eigenvector equation. The above symbolic execution approach can be viewed as an iterative solver for this eigenvalue equation which implicitly exploits the sparseness of the transition matrix. Either way, we now have a tool for quickly screening a number of alternative test loops without having to run the program once the parameters p_{ij} and c_i have been measured for typical input data. Unfortunately, the task of finding an optimal test loop for a given control flow turns out to be **NP**-hard. In order to make this more precise we formulate our problem as a decision problem in the format of [5]:

TEST LOOP SCHEDULING

INSTANCE: Operations $o_1, \ldots, o_n$ and costs $c_i \in \mathbf{N_0}$; transition probabilities $p_{ij} \in \mathbf{Q}$, a test loop length m and a cost bound $C \in \mathbf{Q}^+$.

QUESTION: Is there a test loop of length m for which the Markov chain model predicts a cost $C' \leq C$?

TEST LOOP SCHEDULING is **NP**-hard because it is a generalization of the SUBINTERPRETER SCHEDULING problem to be discussed in Section 4.2. Even for moderate numbers of operations we therefore have to resort to heuristics like hill climbing or genetic algorithms in order to arrive at good test loops.

4 A special case: Interpreter loops

There is quite a number of papers on emulating MIMD behavior (e.g. [2, 1, 8, 3]) which on the first glance are based on a slightly different road to solving the

```
initialization
LOOP
    IF currentInstruction = I_1 THEN execute I_1
    ⋮
    IF currentInstruction = I_n THEN execute I_n
    IF TRUE THEN
        save results
        fetch next instruction
        fetch operands
```

Figure 4: MIMD interpreter as a test loop.

problem: The SIMD machine can interpret a locally stored program written in a RISC like machine language. However, Figure 4 shows that such an interpreter can be viewed as a special case of the general test loop of Figure 2. So far, we have not used this approach because it has a number of problems: In order to limit SIMD overhead there can only be a small number of simple instructions. Accessing instructions and operands requires several indirect memory accesses which are very slow on contemporary SIMD machines. Typically this takes one or two orders of magnitude more time than executing an instruction like add. Finally, many of the operations which are used for the applications described in Section 6 would correspond to hundreds of machine instructions which would have to be interpreted one by one. It is not even clear whether a complex program would fit into the local memory of a SIMD computer.

For all these reasons, a pure interpreter approach cannot be expected to yield practically useful performance on todays machines. On the other hand, interpreters have the conceptual appeal that they can handle arbitrarily complex programs with a fixed number of instruction, whereas the number of operations derived from the control flow of a program can in principle grow without bound. For some applications it might therefore be a good idea to take the best out of both worlds: A small general purpose instruction set for flexibility, and additional coarse-grained instructions specifically tuned for the program to be executed which do most of the real computation.

We now apply the techniques derived in the preceding sections to the interpreter approach. This can also serve as an example how these techniques can be adapted to incorporate other kinds of additional knowledge about control flow.

4.1 Modeling interpreters by Markov chains

In [1] interpreters are modeled using a Markov chain by assuming that instructions are independent and that an instruction I_i can be fully characterized by its probability of occurrence p_i and its cost c_i. This can be viewed as a special case of our Markov model for arbitrary test loops from Section 3. We introduce one operation for each instruction plus one special operation o_0 for accessing instructions and operands (with cost c_0). We know the control flow of the interpreter. An instruction is always followed by o_0 and o_0 is followed by one of the instructions according to their probabilities. All other transitions are impossible.

$$p_{ij} = \begin{cases} 1 & : \quad i = 0 \text{ and } j > 0 \\ p_i & : \quad i > 0 \text{ and } j = 0 \\ 0 & : \quad \text{all other cases} \end{cases}$$

As in Section 2.1, performance can be increased by optimizing the test loop. The special structure of control flow implies that every sensible test loop can be written as $S_1;o_0;S_2;o_0;\ldots;S_k;o_0$ (up to cyclic permutation). Where each S_j is a nonempty subset (called *subinterpreter* in [2]) of the instruction set. Its instructions can be tested for in some arbitrary order. Test loops of a different form would contain tests which can never be successful.

4.2 The NP-hardness of subinterpreter scheduling

The interpreter loops described in Section 4.1 have a considerably simpler structure than general test loops. So, we might hope that there are efficient methods for optimizing them. We now show that this is not the case. Consider the decision version of our optimization problem:
SUBINTERPRETER SCHEDULING
INSTANCE: Instructions $I_1, \ldots, I_n$ and costs $\{c_0, c_1, \ldots, c_n\} \subseteq \mathbf{N}$; probabilities $p_i \in \mathbf{Q}^+$ ($\sum_{i=1}^n p_i = 1$), a subinterpreter count $k \in \mathbf{N}$, a test loop length m and a cost bound $\overline{C} \in \mathbf{Q}^+$.
QUESTION: Is there a test loop with k subinterpreters and $k + \sum_{j=1}^k |S_j| = m$ for which the expected cost per executed instruction is $C \leq \overline{C}$?

Theorem 1 *SUBINTERPRETER SCHEDULING is* **NP***-hard.*

Proof: Consider the well known **NP**-complete partition problem (quoted from [5]):
PARTITION
INSTANCE: A finite set A and a "size" $s(a) \in \mathbf{Z}^+$ for each $a \in A$.
QUESTION: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) \ ?$$

We now transform an instance of PARTITION into an instance of SUBINTERPRETER SCHEDULING:
Let $n = |A|$, $\{I_1, \ldots, I_n\} = A$, $c_0 = 0$, $c_1 = \cdots = c_n = 1$, $p_i = s(I_i)/\sum_{a \in A} s(a)$, $k = 2$, $m = n + 2$ and $C = \frac{3}{4}n$. This is a legitimate instance of the SUBINTERPRETER SCHEDULING problem and it can be constructed in polynomial time. Consider an optimal test loop for this instance. Because $k = 2$, $m = n + k$ and $p_i \neq 0$ it must have the form $S_1;o_0;S_2;o_0$ with $S_1 \cup S_2 = A$ and $S_1 \cap S_2 = \emptyset$. Let $\alpha = \sum_{I_i \in S_1} p_i$. It is sufficient to show that the cost measure C is $\frac{3}{4}n$ if $\alpha = \frac{1}{2}$ and larger in all other cases.

The test loop can be modeled using a Markov chain with the following four states:

$A_{1/2}$: About to interpret S_i and control flow waits for an instruction from S_i.

$W_{1/2}$: About to interpret S_i but control flow waits for an instruction from S_i.

When the Markov chain is in state A_1 we can next book the successful execution of an instruction and the test loop will next be ready to interpret an instruction from S_2. With probability $1 - \alpha$ the control flow will also wait for an instruction from

S_2 now, resulting in a transition to state A_2. In a similar way the other transition probabilities can be found resulting in the following Markov chain:

$$W_2 \underset{\alpha}{\overset{1}{\rightleftarrows}} A_1 \underset{\alpha}{\overset{1-\alpha}{\rightleftarrows}} A_2 \underset{1}{\overset{1-\alpha}{\rightleftarrows}} W_1$$

Markov-chain theory (For details refer to [12]) yields the equilibrium probability to find the Markov chain in either of the active states A_1 or A_2:

$$a = \frac{1}{2(\alpha^2 - \alpha + 1)}.$$

Each traversal of the test loop incurs a cost n and results in two state changes of the Markov chain. Therefore the expected cost per executed instruction is $C = \frac{n}{2a} = n(\alpha^2 - \alpha + 1)$. This cost measure takes the minimum $\frac{3}{4}n$ for $\alpha = \frac{1}{2}$. □

5 Optimal probabilistic test loops

Since assembling optimal test loops turns out to be intractable, it is a logical idea to further simplify the model in order to be able to derive closed form results. We now summarize the results of an approach which is discussed in more detail in [12]. The idea is to abstract from the execution order in the asynchronous control flow and the test loop. Operations are characterized by their frequency p_i of occurrence in the asynchronous control flow. (It can be measured by counting how often an operation is actually executed.) We want to know at which frequency f_i operation o_i should be tested for in order to achieve optimal throughput. This frequency is defined by a probabilistic test loop which randomly decides which operation is tested for next:

LOOP choose i with probability f_i; **IF** g_i **THEN** o_i **ENDLOOP**

It can be shown that the choice

$$f_j = \frac{\sqrt{\frac{p_j}{c_j}}}{\sum_{i=1}^{n} \sqrt{\frac{p_i}{c_i}}}. \qquad (1)$$

yields an optimal probabilistic test loop. Unfortunately it turns out that an optimal probabilistic test loop usually performs worse than even a naive deterministic test loop. Nevertheless, the frequencies calculated by Equation (1) can be used as a hint how many duplications of an operation might be useful. Furthermore, the calculations also yields an estimate for the achievable improvement by operation duplication.

The above result can be modified to accurately model probabilistic test loops for the interpreter case by taking the special role of the operation o_0 into account. The optimal testing frequencies again follow Equation (1).

6 Implementation experiences

We know summarize some experiences with two applications of the concepts developed here to parallel depth first search on SIMD-machines. Section 6.1 is devoted to an algorithm looking for transition tables of a cellular automaton which solves the *firing squad synchronization problem* with a minimum number of states. A more detailed description of the application can be found in [11]. Section 6.2 discusses the SIMD related experiences made with massively parallel game tree search [7].

Both applications have in common that all known data parallel algorithms for them are based on exhaustive search and would be prohibitively inefficient. So the only choice is either to use the technique of emulating MIMD-behavior or not to use a SIMD machine at all. Considering the fact that there are currently no large scale SIMD machines with state of the art technology on the market, the choice for a real production application would probably be "don't use SIMD". But this situation might change again. At the time these applications were developed the 2^{14} PE MasPar MP-1 used for the implementation was the most powerful machine available at the University of Karlsruhe and the SIMD-overhead did not change this situation. Furthermore, there might be applications whose computations are mostly well suited for SIMD-machines but which have a small MIMD-component which might be a bottleneck without our techniques.

6.1 The firing squad synchronization problem

The *firing squad synchronization problem* (FSSP) is a classical problem of cellular automata theory. The task is to find a transition table for a one dimensional cellular automaton which causes all cells at once to enter a "firing" state. One is interested in solutions which work in minimal time and which employ a minimal number of states. We have devised a heuristic (but complete) backtrack search procedure which is able to efficiently search the space of possible transition tables. With this algorithm it was possible to settle the question whether there is a time-optimal solution with four states. The answer is "no".

The FSSP-algorithm cannot directly be translated into an efficient data-parallel algorithm because it consists of three interwoven asynchronous loops for the partial simulation of a cellular automaton, backtracking and a relatively complicated pruning heuristics. Several sets of operations for SIMD emulation of this algorithm have been developed. The final set consists of seven operations whose cost varies by a factor of four and whose relative importance varies by a factor of 16 (with the exception of one very rarely used operation.) This set performs by a factor of about 1.7 better than another operation set it was derived from using some nontrivial instances of the transformation rules described in Section 2.2. By duplicating some of the operations in the test loop and (manually) ordering the operations an improvement by a factor 1.6 in efficiency compared to a naive ordering was possible. It is noteworthy that either duplicating important operations alone or cleverly ordering the test loop yields improvements of only about 10 %. This not only refutes the assertion in [8] that duplication is useless but might also explain the small improvements achieved there by using loop ordering alone.

The control overhead for the test loop is negligible because the state of control can be held in a register and because the operations are rather coarse grained. We expect that an implementation using an interpreter loop would be at least an order of magnitude slower.

Together with an effective dynamic load balancing scheme for distributing subtrees (see [11, 9]) which achieves a processor utilization of more than 80 % and incurs a communication overhead of less than 15 %, the program achieves about 38 times the performance of a sequential implementation on a SPARC-2 workstation.

6.2 Game tree search

This application implements the $\alpha\beta$-heuristics for game tree search together with the "Young Brothers Wait" concept for parallelization described in [4]. It uses synthetic game trees of varying shape in order to answer the question how far the results for up to 1024 PEs achieved on a MIMD-machine can be transferred to exploit even more massive parallelism. For very large search trees with little move-ordering information, speedups up to 5850 were achieved and the algorithm executed 27 times faster than on a 85 MHz SPARC 5. For smaller and more irregular search trees the results are less promising. But the main problem lies in the fact that there are no known parallel game tree search algorithms which are able to exploit massive parallelism without incurring a substantial increase in the number of expanded nodes.

In addition to three operations implementing the sequential $\alpha\beta$-heuristics three communication operations for load balancing, passing results and pruning subtrees have been incorporated into the process of finding an efficient test loop. Choosing the test loop can affect the efficiency by a factor of 2–3. This choice depends on the size and shape of the tree. This is no problem for game tree search because for a given game and search time the size and shape of the trees for subsequent moves are similar. The overall SIMD-overhead can be as small as 22% if the node evaluation function is amenable to a data-parallel implementation.

7 Conclusions

There is no clear-cut border between SIMD algorithms and MIMD algorithms. A program with asynchronous control flow can be decomposed into a number of elementary operations which can emulate asynchronous behavior on a SIMD machine. For many applications, a small number of coarse-grained operations is sufficient resulting in an acceptable emulation overhead. However, more complicated programs may require a large number of operations or the decomposition into very fine-grained operations which resemble a machine instruction set. In this case, the overhead may become prohibitive.

Using the techniques developed here, it is possible to transform a program into a form more suitable for SIMD execution. The transformations can be applied manually but the most important ones are also sufficiently well defined in order to be performed by a compiler. Using a mixture of quantitative and qualitative tools, the emulation can be made considerably more efficient than a straightforward approach.

The most interesting quantitative tools used here are Markov chain models of control flow. They model the behavior of a test loop in a quite general setting and for probabilistic test loops it is even possible to derive closed form expressions for optimal testing frequencies. They also play a key role in proving that finding optimal deterministic test loops is an **NP**-hard problem.

Acknowledgments

I would like to thank S. Egner, H. Hopp, M. U. Mock, R. Vollmar, P. A. Wilsey and T. Worsch for the many interesting discussions which helped to develop and refine the ideas described here.

References

[1] N. Abu-Ghazaleh, P. A. Wilsey, X. Fan, and D. Hensgen. Variable instruction issue for efficient MIMD interpretation on SIMD machines. In H. J. Siegel, editor, *Eight International Parallel Processing Symposium*, Cancun, 1994.

[2] H. G. Dietz and W. E. Cohen. A massively parallel mimd implemented by simd hardware. Technical Report TR-EE 92-4, Purdue University, 1992.

[3] X. Fan, N. N. Abu-Ghazaleh, and P. A. Wilsey. On the complexity of scheduling MIMD operations for SIMD interpretation. *Journal of Parallel and Distributed Computing*, 29:91–95, 1995.

[4] R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 94–103, 1994.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman and Company, New York, 1979.

[6] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. *SIGPLAN Notices*, pages 188–199, 1992.

[7] H. Hopp and P. Sanders. Parallel game tree search on SIMD machines. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 349–361, Lyon, 1995. Springer.

[8] M. Nilsson and H. Tanaka. MIMD execution by SIMD computers. *Journal of Information Processing*, 13(1):58–61, 1990.

[9] C. Powley, C. Ferguson, and R. E. Korf. Depth-first heuristic search on a SIMD machine. *Artificial Intelligence*, 60:199–242, 1993.

[10] P. Sanders. Emulating MIMD behavior on SIMD machines. In *International Conference Massively Parallel Processing Applications and Development*, Delft, 1994. Elsevier.

[11] P. Sanders. Massively parallel search for transition-tables of polyautomata. In *Parcella 94, VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, pages 99–108, Potsdam, 1994.

[12] P. Sanders. Efficient emulation of MIMD behavior on SIMD machines. Technical Report IB 29/95, Universität Karlsruhe, Fakultät für Informatik, 1995.

[13] S. Tomboulian and M. Pappas. Indirect addressing and load balancing for faster solution to Mandelbrot set on SIMD architectures. In *3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 188–199, 1992.