# HIERARCHICAL PARALLELISM EXPLOITATION

Alexandru Nicolau

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

Technical Report No.89-32

# HIERARCHICAL PARALLELISM EXPLOITATION

Alexandru Nicolau

## 1 Introduction

The generation of hand-crafted code for efficient execution on parallel machines is a tedious task. For some important problems, new *algorithms* carefully designed for parallel execution are being developed, often tailored to a particular architecture. However, these algorithms are difficult to develop and implement—the problem must be of sufficient generality, interest and regularity to compensate for the considerable effort. Even when the core algorithms are hand-parallelized, complex *application* codes will not run at large speedups if the rest of the code is not speeded up as well. Furthermore, even the carefully crafted parallel algorithms are likely to contain parallelism that is too low-level and too irregular to be explicitly exploited by the human designer. The remaining parallelism has a multiplicative effect on the overall performance of the code. Thus the ability to exploit parallelism at *all* levels is critical for execution speed.

### 1.1 How Should Parallelism Be Exploited?

Automatic fine-grain (instruction level) parallelism holds the promise of exploiting substantially all the parallelism available in a given program, including highly irregular forms of parallelism not visible at coarser levels. Since the effect of all levels of parallelism exploitation have a multiplicative effect on overall performance, substantially all parallelism should be exploited in order to achieve good performance—an obvious consequence of Amhdal's law. The importance of fine-grain parallelism exploitation has already been recognized to a small extent, and is reflected in the use of pipelining and (relatively narrow) horizontal microcode, in virtually all high-performance (numerical) processors. However, its wider application has been limited by several factors, to be discussed shortly. In this paper we will describe some new results on the exploitation of fine-grain parallelism and will discuss their implications for the design of massively parallel machines.

Ideally, fine-grain parallelism would be exploited at runtime, when all data-dependencies are strict (i.e., there is no ambiguity between indirect references) and a unique execution path through the code is followed. This is essentially the approach taken in the data-flow model of computation. In practice however, the runtime overhead involved in dynamic (hardware) scheduling of operations and interlocking to ensure dependency preservation is often several times larger than the theoretical performance speedup. The alternative approach is compile-time parallelization of the code. The obvious advantage of this approach lies in the elimination

of runtime overheads (by doing the scheduling work at compile-time). This yields simpler, and thus cheaper and faster, machines. Furthermore, this approach can potentially exploit parallelism that is not readily available at coarser levels of granularity, and is far too tedious to be expressed at the user level.

## 1.2 Difficulties in Static Fine-grain Parallelism Extraction

Unfortunately, several difficulties have limited compile-time fine-grain parallelism exploitation. These are:

- Very tight coupling of processors. To achieve maximal benefits, the hardware behaviour should be highly predictable. For example, if processors are synchronous, operations could be executed in parallel in this model, utilizing the "free" implicit synchronization, while the same operations would not be worth executing in parallel if explicit synchronization were required.[1] While this requires high memory bandwidth and constrains the scalability of the architecture, it is technologically feasible, and typically easier to build (and thus less expensive) than complex dynamic interlocking/scheduling hardware. The main drawbacks of the approach are in the ability of the compiler to expose enough fine-grain parallelism to efficiently utilize such hardware.

- Amount of parallelism exploitable by fine-grain techniques. Due to a misunderstanding of some early experiments this was widely (and erroneously) believed to be too small or too expensive to bother with. Later evidence, [21], has conclusively established the availability of rather large amounts of fine-grain parallelism (factors from 10 to 100) in ordinary code.

- Conditional jumps. Since branches occur very often in ordinary programs (once every 3-8 instructions on average), they make the static scheduling of large numbers of operations difficult. Previous techniques have either been limited to branch-free code (basic blocks), thus drastically limiting the potential parallelism, or strongly relied on heuristics to statically predict the direction of runtime branches, with potentially heavy penalties in cases where such prediction is unsuccessful.

[1] This assumes that keeping the processors synchronous is done with negligible cost, and/or does not affect the cycle time significantly.

- Loop limitations. Static fine-grain parallelization was essentially limited to acyclic code (i.e., no loops), although loop-unwinding attenuates this problem to some extent.

- Resource utilization efficiency. Code compacted (parallelized) by fine-grain methods tends to vary a great deal in the amount of parallelism exposed. This variation is particularly troublesome in loops, since each iteration will tend to have a relatively sequential startup and/or wind-down sequence of operations, with a burst of very parallel code in between. This leads to inefficient use of resources and/or degrades performance.

- Unpredictable (ambiguous) data-dependencies. When indirect references occur in a program it can be difficult—or impossible—for a compiler to decide whether two such references do or do not refer to the same memory location. This forces the compiler to make very conservative assumptions about the order in which the instructions have to execute, resulting in slower (less parallel) code.

### 1.3 Overview of the Paper

In this paper we present a set of techniques that combine to overcome all of the above difficulties. We discuss a set of simple *core* program transformations that can expose substantial amounts of parallelism even in cases where alternate techniques would fail. Furthermore, these transformations are defined independently of any superimposed heuristics, yielding increased flexibility. Two additional transformations extend the applicability of the core transformations to arbitrary loops. The first of these meta-transformations allows the exploitation of fine-grain parallelism across multiple nested loops, while the second realizes the *full effect* of complete unwinding of loops, *without* the actual complete unwinding. The transformations work even in the presence of conditional jumps. Together, these transformations combine to overcome all of the loop-related difficulties previously mentioned.

The resilience of our approach to statically unpredictable conditional jumps has been confirmed by both our own experimental evidence, and by independent work at IBM T.J.Watson research Center [10]. In fact, Percolation Scheduling was found to be so robust in the presence of control-flow unpredictability, that its main target application in the IBM project is in systems and casual code domains.

The techniques we present are most effective in the context of synchronous shared memory multiprocessors, although they could be used for asynchronous processors as well. In fact,

we can show that the effect of many previous (coarser grain) techniques (e.g., vectorization, wavefront/hyperplane, loop-interchange, doacross) can be obtained as restricted combinations of our transformation. This provides us with a means of comparing transformations across several computation models. In that context it becomes obvious that the power of the transformations to extract parallelism increases when the target architecture is tightly coupled and synchronous.

We will use our results above to argue that statically scheduled, tightly coupled synchronous architectures are both critical and practical, for the efficient exploitation of massive parallelism. On the other hand, due to hardware issues and other pragmatic considerations (e.g., compilation time, space considerations) it is unlikely that the fully static approach will directly scale up to massive ( $\cdot$ 1000) parallelism exploitation. Fortunately, since good programming techniques tend to yield structured (hierarchical) code with relative locality, tight coupling and static scheduling at the higher levels of the hierarchy (e.g., across procedures/modules) become less important—the ratio of synchronization/communication across processors decreases relative to the code size). This leads to the notion of a general interconnection network with each node consisting of a set of (possibly dynamically partitionable) tightly coupled *synchronous* processors.

# 2 Compile-time Fine-grain Parallelism Extraction

In this section we discuss the tools necessary for exposing parallelism in ordinary programs from the (machine) instruction level up to the procedure level. For the purposes of the section we assume that the hardware on which the code will ultimately run efficiently supports this granularity of parallelism. As we have argued in the introduction, such support is critical since *all* levels of parallelism need to be addressed to "beat" Amhdal's law. In the next section we will discuss the practicality of such an architecture.

## 2.1 Analysis Tools: Disambiguation

A large fraction of the parallelism available in programs involves indirect references. Thus, it is imperative for a parallelizing compiler to be able to effectively disambiguate as many indirect references as possible. Indeed, too liberal an approach to disambiguation could result in incorrect code being generated, while too conservative an approach will sharply decrease

```
   for i = lb,ub do
       j := 2*read(); /* read() reads an integer from the standard input */
(a)    A[2i+1] := expr1;
(b)    B       := A[j];
       od;
```

Figure 1: Sample Problematic Ambiguity

the effectiveness of the overall parallelizing compiler. For example, figure 1 shows a situation in which a simplistic (and conservative) compiler, will assume that statements (a) and (b) have to be executed sequentially. Note that this is *the only* safe assumption that the compiler can make, unless it has the ability to ascertain that the references in (a) and (b) cannot refer to the same memory location. Even in this trivial example, the ability to disambiguate the two references in (a) and (b) could result in a significant speedup, by allowing (a) and (b) to execute in parallel. It should therefore be intuitively obvious that the ability to perform accurate disambiguation of indirect references is crucial for a parallelizing compiler. We will shortly present quantitative evidence to this effect for the Bulldog compiler [12].

## 2.2 Performing Static Memory Disambiguation

Memory disambiguation techniques for determining (to the extent possible based on *fully-static*[2] information) whether two indirect references might access the same memory location can be found in [7], [22], [19]. They involve the derivation of primitive expressions for the array indexes. These expressions contain compile-time constants, and variables whose values cannot be derived at compile-time. To determine whether two references conflict, the primitive index expressions are symbolically equated and the resulting *diophantine* equation is solved. If there are (integer) solutions to the equation, then the two references might access the same memory location, and a potential conflict (data-dependency) must be assumed. For example, the indices in statements (a) and (b) in Figure 1, would first be analyzed and transformed into a canonical form consisting of only constants and irreducible variables. In particular, $i$, the induction variable of the loop is transformed, if necessary, into a function of the canonical

[2] Fully static information is knowledge about the program that can be completely derived from facts known at compile-time.

loop induction variable, say $l_1$ [3]. Since $j$ involves an input variable no further reduction is possible, and $j$ is expressed as $2 * r$ by the disambiguator. Then the diophantine equation:

$$2l_1 + 1 = 2r$$

is solved, using techniques derived from standard number-theory. Since in this particular case no integer solutions exist, the compiler can *safely* assume that no conflict can occur between the two statements. Thus they can be executed in any order, and in particular in parallel.

## 2.3 Effectiveness of Static Disambiguation

Indirect references in inner loops of scientific code are mostly array references, and such code usually offers the greatest potential for parallelism. Thus the very accurate disambiguation of indirect references is crucial to the success of fine-grain parallelizing compilers.

Evidence supporting both the effectiveness of disambiguation and its importance for a fine-grain compiler is provided by our experiments with the BULLDOG compiler. Table 1 compares the results obtained by the BULLDOG compiler with and without its (fully-static) disambiguator system, for several programs and various unwindings. Even with the limited unwinding used for some of these tests[4] the importance of disambiguation becomes obvious. The significance of disambiguation for the performance of the compiler increases dramatically with larger unwindings.

The programs (a fast Fourier transform, solving a system of linear equations, tridiagonalization, matrix multiplication, finding prime numbers and transitive closure) are all dramatically improved by the use of the disambiguation system; the speedup is essentially doubled in several cases by the disambiguation. As expected, the improvement is particularly large when the traces are long and the potential speedups obtainable by trace scheduling are relatively large. This happens when the important (innermost) loops are unwound. When unwinding is not done, or traces are still small, the length of the compacted schedule is dominated by simple arithmetic dependency-chains (e.g., index calculations may determine the length of the trace schedule) and no large speedups will be achievable in any case. Under these conditions the effect of disambiguation decreases.

[3] In general this further improves the accuracy of the disambiguation process by eliminating multiple induction variables in a loop. Such variables would otherwise become free variables in the diophantine equation.

[4] The number at the end of the program names indicates the amount of unwinding.

Table 1: Effectiveness of fully-static disambiguation.

| *Program* | *Instructions produced W/O Disambiguation* | *Instructions produced W/ Disambiguation* |
|---|---|---|
| FFT1 | 1215 | 1135 |
| FFT4 | 869 | 521 |
| FFT16 | 800 | 415 |
| SOLVE1 | 10007 | 10007 |
| SOLVE4 | 8625 | 5169 |
| TRID1 | 2988 | 2988 |
| TRID8 | 2469 | 1401 |
| MATMUL1 | 109 | 109 |
| MATMUL4 | 53 | 41 |
| PRIME1 | 656 | 656 |
| PRIME4 | 427 | 321 |
| TRCL | 78 | 53 |

Examination of the disambiguation results reveals that the speedups could be even more dramatic, if the—already considerable—effort the disambiguator puts in analyzing the code were further increased.

## 2.4 Limitations of Fully-static Disambiguation

While a large fraction of indirect references occurring in ordinary code (particularly scientific code) are array references that can be disambiguated at compile-time, others are too dependent on runtime information to yield to any fully-static analysis. For example, the code in Figure 2 (unlike that in Figure 1) cannot be disambiguated by *any* purely static analysis. Also intractable by fully static analysis are most pointer references, and multiple indirection (e.g., scatter/gather).

In practice, the disambiguator system will be limited in its ability to successfully disambiguate references not only by ambiguities fundamentally intractable at compile time, such as the above, but also by the system's own imperfections. For example most disambiguators will not deal well (if at all) with division, general non-linear indices, or interprocedural analysis. Similarly range analysis [13] and control-flow analysis are usually sacrificed to improve the running time of the compiler. All of these weaknesses tend to decrease the performance of

```
      for i = lb,ub do
         j := read();  /* read() reads an integer from the standard input */
(a)      A[2i+1] := expr1;
(b)      B       := A[j];
         od;
```

Figure 2: Ambiguity not Handled by Fully-Static Disambiguation

the dependency analysis tool.

## 2.5 What Runtime Disambiguation Has to Offer

What we have proposed is the shifting of part of the burden of disambiguation from compile-time to runtime. While the scheduling decisions will still be made statically, they may occasionally rely on runtime tests to guarantee correctness. This will have the advantage that the disambiguation information rather than having to be *always right* (i.e., verifiable) *statically*, would only need to be *usually* (or often) right. This relaxation allows—in principle—the disambiguator to handle all of the above cases, and in fact could be used not only as a complement for a fully static disambiguator, but even—within limits—could make up for the lack of sophisticated—and slow—fully static disambiguation. Thus the use of runtime disambiguation could dramatically improve not only the running time of the code generated but also the running time of the compiler itself.[5]

This approach is new for parallelizing compilers. Previous techniques relied exclusively on fully-static information to estimate data-dependencies. This conservative approach, discussed in the previous section, has undully restricted the effectiveness of parallelizing compilers. In fact, some of the chief critiques voiced against parallelizing compilers (e.g., [14]) center precisely on the *perceived* intrinsic need of such compilers to rely solely on fully-static analysis, and their resulting inability to exploit the "real" parallelism limited only by actual runtime dependencies. Runtime Disambiguation (RTD) comes to remedy this problem of parallelizing

[5] Between 1/3 and 1/2 of the running time of the Bulldog compiler [12] is spent in preparing accurate fully-static dependency information. Even with all this effort, the compiler still missed some relatively simple—and important—disambiguations. An assertion facility was added to the system precisely to allow the user to overcome such problems.

```
(a)  A[i]
     .......

(b)  A[j]
```

Figure 3: Original code before RTD code insertion

```
(a)          A[i]
            ..........
             if i =\= j  l1 l2 .9 /* i not equal j */
         l1: assert i =\= j
(b)      l2: A[j]
```

Figure 4: Code after RTD code insertion

compilers, while at the same time still avoiding most of the dynamic overheads inherent in pure runtime approaches.

The RunTime Disambiguation system (RTD) we have implemented treats memory anti-aliasing of references which cannot be effectively disambiguated at compile time. Using it, part of the disambiguation mechanism is integrated into the parallel code produced by the compiler.

Given two references that cannot be disambiguated at compile-time, a potential conflict between them (i.e., $i = j$) (Figure 3) has to be conservatively resolved to ensure correctness using the traditional disambiguation approach. RTD on the other hand, will transform the code segment to that in Figure 4. The .9 probability estimate gives the trace (a,l1,b) priority in the compaction process. The assertion in statement l1 will be placed in a small data-base for the disambiguator and will supercede any information the disambiguator may have obtained about the references for that path. The asserted information will be used as a compiler directive, allowing a,b to be scheduled independently. Of course, the off-trace branch must take care *at runtime* of the case in which $i = j$. Depending only on the computation of $i$ and $j$, the *if* statement might be placed early in the schedule (quite possibly for "free" if

resources are available). So if the conflict between the references is either nonexistent (but the disambiguator cannot establish that by itself at compile time) or occurs rarely (e.g., $i$ and $j$ are used to traverse A in opposite directions), the potential speedup resulting from the use of RTD can be significant, as shown in table 2. RTD can also be used in the form of dynamic assertions, i.e., the user introduces assertions about the absence of dependencies which allow the compiler to parallelize the code as if no dependency was present, but also result in the insertion of a runtime check and escape to guarantee correctness. This feature can be very useful, by allowing good parallelization in the presence of transient (occasional) dependencies, or boundary conditions; it can also provide a fool-proof assertion mechanism for relatively naive users. More details on the implementation of RTD, as well as a detailed analysis of its performance are found in [18].

## 2.6 Instruction-level Parallelism Extraction

Existing compilers for parallel machines do not provide the needed support. While important advances in the parallelization of ordinary code (especially vectorization) have been achieved [15], [5], there is still a lack of satisfactory tools to automatically extract fine and coarse grained parallelism in a unified fashion. The goal of Percolation Scheduling (PS)—our hierarchy of fine-grain code transformations—is to provide such tools. Thus PS can be thought of as complementing and/or enhancing previous approaches to parallelism extraction.

PS globally rearranges code past basic block boundaries in an attempt to gain parallelism. Its core is a small set of primitive program transformations defining the allowable motions of operations between adjacent nodes in a *parallel program graph*. A parallel program graph is a directed graph in which each node contains one or more operations that can be executed in parallel. The edges in the program graph determine the execution-paths in the program. The goal of PS is, then, to maximize parallelism by moving operations from node to node so as to maximize the number of parallel operations in the final graph. A precise definition of the execution semantics is found in [20]. PS core transformations are easy to understand and implement. Furthermore, they are atomic and can be combined with a variety of guidance rules to direct the optimization process. Above this core level are guidance rules and transformations which extend the applicability of the core transformations and exploit coarser parallelism. The main advantage of PS over previous approaches, is its resilience to unpredictable control flow, its modularity, and its uniformity of application.

Table 2: RTD Net Speedups.

| *Program (unwinding)* | *Speedup RTD vs NoRTD* | *Speedup AU vs NoRTD* | *Speedup TR vs NoRTD* |
|---|---|---|---|
| dotprod(4/8) | 0/0 | 0/0 | 0/0 |
| ln(4/8) | 0/0 | 0/0 | 0/0 |
| matmul(4/8) | 0/0 | 0/0 | 0/0 |
| sqrt(4/8) | 0/0 | 0/0 | 0/0 |
| Conduc(4/8) | .15/.22 | (< 0)/.10 | .15/.20 |
| FFT(4/8) | .26/.42 | .10/.34 | .26/.42 |
| Trid(4/8) | .18/.23 | (< 0)/.20 | .18/.23 |
| Quanc(4/8) | .15/.28 | .08/.22 | .15/.26 |
| SVD(4/8) | .26/.44 | .15/.40 | .24/.41 |
| Solve(4/8) | .16/.27 | .10/.27 | .16/.27 |
| Invert(4/8) | 2.2/3.4 | .5/.9 | 2.2/3.4 |
| BinSort(4/8) | 3.7/7.1 | 2.3/5.1 | 2.5/5.3 |
| BubleSort(4/8) | .7/.9 | .4/.5 | .4/.6 |
| ShellSort(4/8) | 1.5/2.4 | 1.2/2.3 | 1.3/2.1 |
| RadixSort(4/8) | 3.2/6.2 | 2.3/5.2 | 2.2/5.0 |
| prime(4/8) | .5/.9 | .3/.7 | .4/.7 |
| trcl(4/8) | 1.3/2.1 | .8/1.7 | .8/1.5 |
| Unions(4/8) | 1.6/3.1 | .7/.6 | 1.3/2.4 |
| Inserts(4/8) | 2.9/4.6 | 1.7/1.9 | 2.4/4.1 |
| ShortesPaths(4/8) | 1.2/2.1 | .9/1.9 | .8/1.6 |

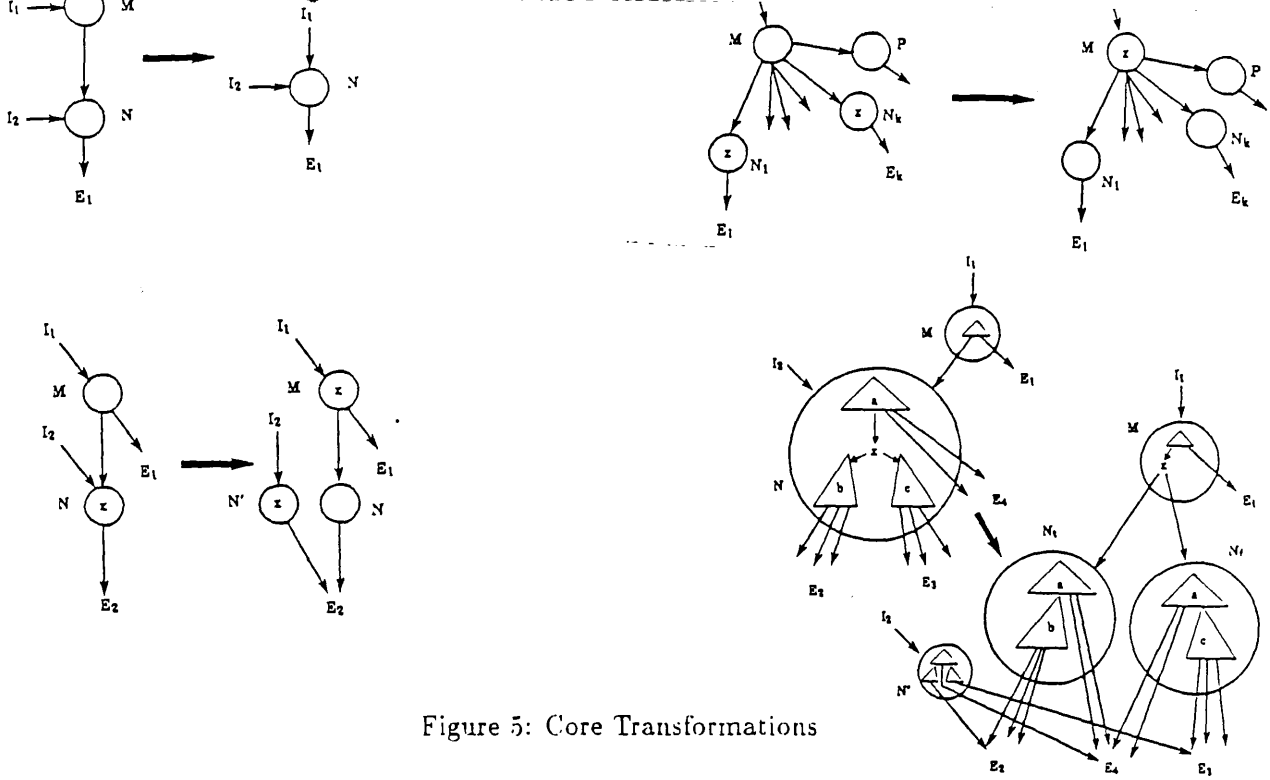


Figure 5: Core Transformations

Guided by the higher level rules and transformations, the core transformations operate uniformly on an entire program graph. They can also be applied to partially parallelized code. This allows modification of code produced by other types of compilers. In addition, these transformations are themselves highly parallel and could be run on a parallel machine, significantly reducing compilation time.

The following is an outline of the layers of the PS system and their function:

**Core Level** This level contains a set of four core transformations that define semantically correct motions of operations between adjacent nodes in a program flow-graph. By "percolating" operations that can execute in parallel to the same node of the graph, the core transformations expose parallelism implicit in the code. These transformations apply directly to loop bodies and non-loop code. They serves as the main parallelization tool in our system. The core transformations are illustrated in Figure 5.

**Support Level** At this level we have analysis methods (e.g., Memory Disambiguation [19]) and standard optimizations (e.g., Dead-Code Removal). They provide accurate data-dependency information and thus enhance the applicability of the core transformations.

**Guidance Level** This level consists of rules that direct the application of the core transformations to achieve effective optimization of the code in acceptable time and space. This contrasts with Trace Scheduling [11] where a single rule (for trace picking) is inseparable from the actual transformation mechanism. This limits Trace Scheduling and makes it too rigid for our goals.

**Enabling Level** This level consists of transformations that allow the core transformations to process arbitrary graphs and enables them to exploit coarser grained parallelism (e.g., within inner and nested loops).

**Meta Level** This level consists of transformations that use the core transformations to exploit coarse parallelism (e.g., partial loop/module overlapping).

The separation of levels yields more general, cleaner transformations. It simplifies both the understanding and the implementation of a PS compiler. Furthermore, the applicability of the transformations is enhanced, and it is easier to experiment with various high level transformations without affecting the correctness of the compiler.

## 2.7 Static Inner Loop Parallelization

Recently, techniques have emerged that produce absolute time-optimal parallel schedules for loop execution on synchronous multiprocessors, subject to the data-dependencies of the loop and the availability of enough resources (i.e., processors) to accommodate the schedule.

The most general technique, Perfect Pipelining (PP) [4], combines the benefits of fine-grain parallelism (can exploit irregular forms of parallelism) with the pipelining of iterations of coarser methods [9]. Perfect Pipelining uses incremental unwinding and successive applications of parallelization (compaction) transformations (e.g., Percolation Scheduling [20]), to detect a *pattern* in the code—which in practice will emerge after a small amount of unwinding. The loop body can then be replaced by this pattern yielding a *schedule* for the loop. It can be shown that given enough resources, and subject to the given compaction transformations, the loop thus obtained will yield the best (optimal) running time, (i.e., further unwinding and compaction of the loop cannot yield better speedups). In particular, the running time of the new loop is identical to what might be obtained by full unwinding of the loop and full fine-grain parallelization, would such unwinding be feasible. This is important, since in practice loops can (or should) seldom be fully unwound at compile-time. These results hold even in the presence of conditional jumps and multicycle operations.

The second technique, Optimal Loop Parallelization (OPT) [3], deals with loops containing no conditionals, or in which conditionals are if-converted [6] (or the probability of paths execution is predictable). For such loops, OPT, which is a refinement of perfect pipelining, can be used to achieve an even stronger result. We can show that given *any* parallelization transformations that preserve the original data-dependencies, our transformation achieves

equal or better running time for the final loop. In other words, OPT not only yields the best running time for the loop with respect to unwinding and the particular parallelizing transformations used, but true time optimality with respect to any possible dependency-preserving transformations.[6] OPT relies on the fact that only a finite (and in practice, small) number of iterations ever need to be examined to determine a pattern which yields an optimal running-time schedule for the loop. These results hold in the presence of multicycle operations. The justification of these claims and the details of the algorithm are given in [3]. For the purpose of this paper we only need to understand how OPT works. OPT incrementally unwinds the loop, allowing operations to be scheduled as early as possible in the schedule, subject only to data-dependencies and latencies.[7] Thus operations are scheduled at the earliest possible time they could be issued at runtime, if a synchronous multiprocessor were available. A repeating, fixed size pattern is guaranteed to emerge after a relatively small amount of such unwinding and compaction (parallelization), if the original data-dependencies of the loop are not allowed to drastically change throughout the process. Further unwinding and compaction beyond this point cannot improve parallelism, and thus replacing the loop body with this pattern will yield an optimal execution schedule for the given loop. Of course, a prolog and postlog including some start-up and wind-down code may be required; this code consists of partial iterations (loop bodies) at the beginning and end of the loop. There are several ways for handling these and other details such as the loop overhead, with either software or hardware support. Some hardware mechanisms which would be relevant have been implemented and are discussed in [8], [10].

An illustration of the effects of OPT and the optimal schedule produced for the given loop is found in Figure 6. For simplicity, latencies of operations in this example are assumed to be just one cycle. As we mentioned earlier, OPT can deal with realistic operation latencies.

When taking into account true operation latencies the notion of optimality derived from OPT/PP is realistic, in the sense that a schedule produced by OPT or PP could be run "as is" on a synchronous parallel machine (e.g., Multiflow's [17]) with enough resources. Still,

[6] Dependency changes (e.g., due to renaming) can be allowed in this context, even if done dynamically as part of the parallelization process.

[7] We are essentially performing a topological sort, creating a partial ordering of the operations; operations that are placed at the same level in the schedule are therefore independent of each other and can be executed in parallel. Given a synchronous parallel processor with enough resources, such a schedule could run "as is", with each level or slice of the schedule issuing each cycle.

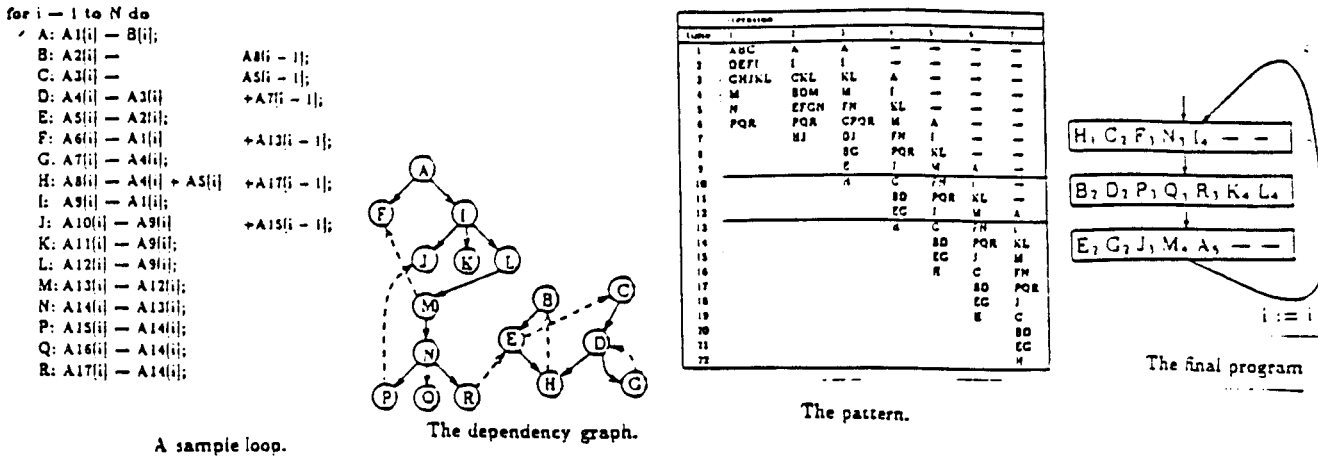



Figure 6: Sample OPT scheduling

in practice, resources are often not available to allow the direct execution of the optimal schedule, and thus a *mapping* phase that adapts the optimal schedule to the actual hardware is needed. The more idiosyncratic the hardware (e.g., structural hazards, non-uniform pipes), the harder this mapping becomes.

In this context, it is natural to ask what the relevance of the optimal schedules is for practically feasible machines. A formal discussion of these issues is found in [1]. In this paper we will further address this topic after considering the issues involved in architectural design for uniform parallelism exploitation.

## 2.8 Nested Loop Parallelization

Loop unwinding has been long known as an effective way to increase the efficient utilization of pipelined machines. More recently loop unwinding has emerged as a primary technique for exploiting fine-grained parallelism within loops, notably for Very Large Instruction Word machines, a form of very tightly coupled multiprocessors [12]. Percolation Scheduling and Perfect Pipelining also use (incremental) unwinding in deriving the optimal pattern for a loop body. While these techniques (PS, PP) are very effective in extracting parallelism inside and outside single loops, sometimes the parallelism in the code may be distributed across several nested loops, so dealing with only one of the loops will not achieve the best results. This situation occurs frequently and is illustrated in figure 7a. On the other hand, arbitrary

unwinding of multiple loops may violate correctness, and thus appropriate checks are needed to ensure that the transformation preserves the semantics of the original code. For example, in figure 7, a 3 by 3 unwinding on each loop would yield an incorrect program.

Loop Quantization is a technique that we have developed to overcome this problem by allowing correct multiple-loop unwinding for arbitrary nested loops. In the case above, for example, the 3 by 3 unwound loop body (the "quantum box") can be *slanted* to become parallel with the dependencies in the code, thereby restoring correctness. Of course the loop bounds need to be modified accordingly to allow for such slanted quantization. In [2] we have shown how the decision on the bounds of Quantization, and the ensuing transformation of the loop, can be automated.

Loop Quantization rearranges the order of execution of the loop iterations less than some other global transformations. For example, quantization will succeed even when straight loop interchange would not apply. By exposing even irregular fine-grain parallelism, quantization may help achieve significant speedups in ordinary code. The main loop of weather code, for example, is naturally amenable to quantization, as are the Livermore loops[16] in their nested context. LQ combines with PP to achieve optimal parallel schedules (for a given number of processors) for nested loops.

An example of loop quantization is given in figure 7; further details and an algorithm for computing maximal loop quantizations is given in [2].

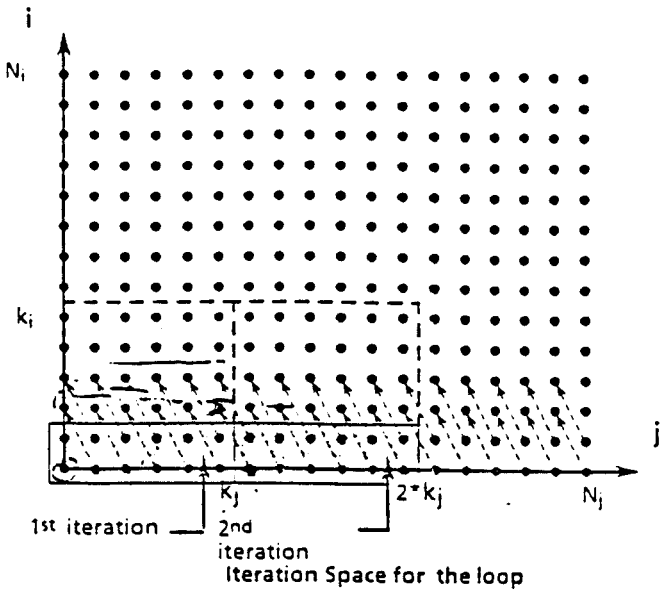
# 3 Architectural Considerations

The above compiler techniques combine to effectively expose virtually all fine-grain parallelism obtainable at compile-time. The techniques are resilient in the presence of unpredictable conditional-jumps, and indirect references. To take full advantage of the potential of these compiler techniques, synchronous multiprocessors are required. While on a small to medium scale (up to a few tens of processors) such machines are relatively easy to build and can be very cost effective—as demonstrated by commercial machines such as [17], [8]—on a larger scale they may involve a number of disadvantages.

## 3.1 Disadvantages of Statically Scheduled Multiprocessors

The main disadvantage of static architectures is that they can't scale up arbitrarily due to:

```
For i := 1, Ni do
  For j := 1, Nj do
    X[i,j] := expr(X[i + 2, j-1]);
    od;
od;
```




(a) Original loop (b) Iteration space and quantum box ;

Figure 7: Loop Quantization.

- Clock drift. as the machine grows larger, it becomes harder to keep all components synchronous. While a synchronous *model* can be enforced by various (hardware software) synchronization protocols, it will ultimately lead to slower cycle times (so as to allow the synchronization of the hardware).

- Memory bandwidth. As more operations are executed in parallel, more simultaneous loads need to be executed. Since memory access is intrinsically sequential, and cannot even be pipelined directly, this implies that the required bandwidths can only be achieved by extensive banking of the memory, coupled with either software or hardware mapping of data to memory banks (e.g., hashing), in an attempt to minimize bank conflicts. To ensure good average performance, the banking extensive even for relatively small (e.g., less than 32 processors) machines, could become unmanageable for vastly larger number of processors. Furthermore, any bank conflict will lead to a delay of all the operations being issued executed: to preserve the synchronous model of execution, the global clock must be frozen until all memory conflicts are resolved: otherwise, complex interlocking hardware would have to be provided, which would again be either hard to build for any large number of processing elements, or would cause significant bottlenecks.

- Conservative scheduling assumptions. As the number of processing elements working in parallel increases, we have to find more opportunities of exploiting parallelism in the code. We have demonstrated that at the nested loop level (and below) enough parallelism exists and can be extracted effectively; a tightly coupled machine of medium size (somewhere between ten to one hundred processing units, is typical of ordinary code in our experience), can make the most of this parallelism. While conservative decisions may sometimes be made at this level to ensure correctness of execution, there is usually no alternative: dynamic mechanisms (e.g., for dependency testing) that are general enough to allow the exploitation of significant amounts of parallelism, are usually too expensive at this level. However, as we go beyond such numbers of processors, and examine coarser levels of parallelism, (e.g., at the procedure level), we may well be slowed down by the conservative decisions implicit in the fully static approach more than by the use of dynamic synchronization.

Out of these obstacles to scalability, the last one is probably the most critical. For example, consider the task of inserting a sequence of elements into a binary search-tree. Two successive calls to *Insert* could clearly execute in parallel as soon as it is determined (at runtime) that the subtrees they need to insert into are disjoint. A purely static approach, however, would need to schedule them for sequential execution, since a conflict could sometimes exist. Of course, RTD could be used to overcome this problem, but that would involve some overhead plus some code duplication for the case where the conflict does indeed arise at runtime. To the extent that the overhead involved in explicit synchronization between the two procedure calls compares favorably with that of RTD, it would be preferable for the user to insert some synchronization code between the calls, (e.g., through the *future* mechanism proposed in Multilisp), and allow for fully dynamic synchronization. The point is, that when the execution time of the tasks (insert) is large relative to the cost of synchronization, the overhead on an asynchronous machine becomes tolerable, and possibly preferable to static scheduling.

More importantly, the parallel execution of multiple procedure calls with their individual threads of control flow implies, in itself, a combinatorial explosion in code size—if encoded in the statically scheduled model. Such an explosion results from the need to encode all possible

paths through the code into a single thread of control, required by the fully static approach[8]. While a single thread of control is a natural paradigm for the execution of non-loop code, individual loops and even most procedure bodies, it obviously becomes impractical at the procedure/module level. While expansion in line of procedures can alleviate this problem, it will not eliminate it.

At this level the tasks are often large and relatively independent of each other—after all, modularity, one of the main reasons for using procedures, discourages extensive sharing of data between procedures and thus minimizes need for communication. Even in cases when communication occurs frequently at the procedure level, it is often of the producer-consumer type. Such unidirectional communication can be effectively pipelined (given that the communication and routing are handled by dedicated hardware operating concurrently with the processors). This will tend to further decrease the relative cost of dynamic synchronization at this level. Also, and perhaps most importantly, at this level of granularity, the user may be better able to share in the burden of parallelism extraction. In fact there is not much choice: if massive parallelism is to be effectively exploited, all components of the program development chain must cooperate. Fortunately, many algorithms that could benefit from massive speedups have an intrinsically parallel high-level structure.

## 3.2 Ideal Architecture

In the light of the fact that good techniques for extracting fine-grain parallelism exist, and that this parallelism can be consistently exploited most effectively by a synchronous machine, it is reasonable to suggest that such a machine should be used as the basic block for any massively parallel architecture, since otherwise an important multiplicative speedup factor could be lost. On the other hand, we have also shown that there are several obstacles in using the fully static approach as the model for a scalable parallel architecture. Thus , we believe a hierarchical architecture, with fine-grain, statically scheduled clusters interconnected by some interconnection network (e.g., a hypercube) is the best approach. The trade-off between the cluster size and the overall size of the machine, remains to be determined empirically. Obviously, the communication speed between clusters should be minimized either by the use

[8]Typically, static machines only allow a single thread of control with possibly a multiway jump mechanism. While not impossible in principle, multiple threads of control on a synchronous static machine would be hard to support.

of dedicated and fast communication processors, or by latency avoidance schemes such as that used in Burton Smith's Horizon.

Such a machine could use heuristic high-level scheduling algorithms as in [23], to map the parallelism exposed at the language (and algorithm level) onto clusters. The effectiveness of this approach is illustrated in [23], where the automatic mapping was shown to be better than that derived by human experts.

### 3.3 Cluster Architecture

Synchronous processing elements, each able to accept (any) one operation per cycle, with operation execution pipelined over multiple cycles. While the number of cycles required to execute a n operation is fixed for each operation type. This presents no particular problem for a register-to-register instruction set, with explicit load/stores. The fixed execution time for loads can be enforced—as far as the processors are concerned—by freezing (all) the processors in the cluster if loads do not complete in the expected time. Alternatively, latency may be masked by trading off parallelism as in the HEP or the Horizon.

Such synchronous processors are obviously buildable on a small to moderate scale (2 to 30 processors), as illustrated by the products of Multiflow, Cydrome, Chopp, FPS. Thus the only other difficulty is in providing a "clean" machine, i.e., free of structural hazards, so that any operation can be accepted by each processing element every cycle. The availability of clean pipes is not crucial to our approach. However, while structural hazards (i.e., irregularities in the machine design that optimize hardware utilization) can reduce the cost of the hardware, the trend in architectural design is to avoid structural hazards as much as possible—clearly any machine with too many structural bottlenecks cannot perform at or near its peak regardless of the compiler technology used. We are arguing that the added (hardware) cost of avoiding structural hazards is now even more justified by the existence of software techniques capable of generating optimal code for clean machines for large classes of loops, and provably good code for the cases where optimality is unfeasible—see bellow. If structural hazards are not completely avoided, then simple techniques such as further unwinding of the OPT schedule and compaction (parallelization) coupled with reasonable mapping algorithms can minimize the impact of the hazards on the quality of the code. This, coupled with the relative simplicity and uniformity of application of OPT/PP makes it a good candidate even for existing pipelined and synchronous parallel machines.

The only other problem is the register-bank bandwidth. A clean architecture needs to support writes per cycle equal to the number of different operation-sizes supported by the machine, and reads equal to the maximal number of operands read per operation. There are several ways to implement this; trickier (but still feasible) for multiple pipes. In any case, loads/stores need to write in separate register bank, since they may have some (unpredictable) variation in completion time, and thus we can't guarantee non interference with other operations. Reads are done globally from all register banks (only two most likely), writes can be guaranteed (by the compiler) to be non-conflicting—simple enough: destination of operations that can finish simultaneously have to be in different banks. Even simpler hardware: each type of operation writes into a different (and unique) register-bank. This may result in either extra moves, or some code duplication on occasion, to get data in the right place at the right time. This can also be done under compiler control.

Thus such machines are feasible, and many of the ideas necessary have already been developed and even incorporated in commercial products. We believe the hardware overhed involved in building a clean machine compares favorably with the hardware overheads existing in other, common architectures, both in terms of complexity and performance implications, and they are more than compensated for by the simplification of the scheduling problem that can potentially lead to better code, hence enhancing speed. Furthermore the cost of the added hardware needed to eliminate hazards is offset by the elimination of the need for hardware interlocks (work done by compiler), which in turn may allow a faster cycle time.

## 3.4 Mapping

In [1] we have shown that the problem of generating optimal code for a single pipelined machine with a unique structural hazard is NP-Complete.

On the other hand, we have shown that the idealized schedules produced by our fine-grain techniques can be adapted to produce optimal code for hazard-free single or multiple processors, for recurrence-free loops. Furthermore, we have shown that even in cases where optimal machine schedules are not achievable, the optimality of the idealized schedule leads to a tight bound on the overall performance, and good empirical performance. If structural hazards are introduced, we show that the schedule may be suboptimal even when an optimal schedule exists for an equal size hazard-free machine. We have also shown that in such cases, an optimal fixed-size schedule for the loop—i.e., one whose size is not a function of the number

| Loop | Original Code | Limited Processors | | Ideal Schedule | | |
|---|---|---|---|---|---|---|
| | Mflops | 1 proc Mflops | 2 procs Mflops | procs | registers | Mflops |
| LL1 | 9 | 31-50 | 57-100 | 13 | 82 | 400 |
| LL2 | 8 | 20-35 | 40-60 | 16 | 105 | 320 |
| LL3 | 7 | 16-20 | 20-23 | 5 | 8 | 27-40 |
| LL4 | 6 | 16 | 20 | 5 | 8 | 27 |
| LL5 | 6 | 12-15 | 15-16 | 3 | 5 | 16 |
| LL6 | 8 | 6-16 | 6-20 | 5 | 8 | 6-27 |
| LL7 | 20 | 36-51 | 71-99 | 36 | 243 | 1280 |
| LL8 | 11 | 40-55 | 80-110 | 60 | 363 | 2400 |
| LL9 | 17 | 35-49 | 68-97 | 39 | 264 | 1360 |
| LL10 | 10 | 18-25 | 36-48 | 40 | 210 | 720 |
| LL11 | 4 | 4-9 | 4-11 | 4 | 4 | 4-13 |
| LL12 | 4 | 13-20 | 27-40 | 6 | 37 | 80 |
| LL13 | 4 | 11-12 | 22-24 | 50 | 376 | 560 |
| LL14 (avg) | 4 | 14-18 | 25-31 | 28 | 161 | 270 |
| Average | 8 | 19-28 | 35-50 | | | 534-537 |
| Harmonic Mean | 7 | 13-20 | 18-33 | | | 25-53 |

Table 3: Cluster Sample Performance on Livermore Loops

of iterations in the loop—is not obtainable in general, so good heuristic performance is all we may expect (and do in fact achieve) in practice. Some sample measurements based on the Livermore Loops [16] are shown in Table 3. The timings of the operations are assumed to be those of the Cray-1. It is interesting to note that the single (pipelined) processor mean performance on these loops is by itself slightly better than that of the Cray-1, while for the two processor version, the performance improves even further. The ultimate performance of a statically scheduled cluster will depend of course on the number of processors, as well as on the actual hardware implementation. More details are to be found in [1].

## References


[1] K. Pingali A. Nicolau and A. Aiken. Fine-grain compilation for pipelined machines. *Accepted for publication in the Journal of Supercomputing, to appear August 1988.*

[2] A. Aiken and A. Nicolau. Loop Quantization: an analysis and algorithm. Technical Report 87-821, Cornell University, 1987.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation.* June 1988.

[4] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*. Springer Verlag Lecture Notes in Computer Science no. 300, March 1988. Also available as Cornell Technical Report TR 87-873.

[5] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. Technical Report MASC TR 82-6, Rice University, 1982.

[6] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 1983 Symposium on Principles of Programming Languages*, pages 177–189. January 1983.

[7] U. Banerjee. *Speedup of Ordinary Programs*. PhD thesis, University of Illinois at Urbana-Champaign, October 1979. 79-989.

[8] Cydrome Inc., Palo Alto, Ca. *Technical Summary*, 1987.

[9] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[10] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pages 69–79, December 1987.

[11] J. A. Fisher. *The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources*. PhD thesis, New York University, 1979.

[12] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 37–47, June 1984.

[13] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3:243–50, May 1977.

[14] R. W. Heuft and W. D. Little. Improved time and parallel processor bounds for Fortran-like loops. *IEEE Transactions on Computers*, C-31(1), January 1982.

[15] D. Kuck, R. Kuhn, , B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of the 4th Int'l Computer Software and Applications Conference*, pages 709–715, October 1980.

[16] F. H. McMahon. Lawrence Livermore National Laboratory FORTRAN kernels: MFLOPS. Livermore, CA., 1983.

[17] Multiflow Computer Inc., Branford, Connecticut. *Technical Summary*, 1987.

[18] A. Nicolau. Runtime disambiguation: Coping with statically unpredictable dependencies. *Accepted for publication in IEEE Transactions on Computers, to appear Fall 1988.*

[19] A. Nicolau. *Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers.* PhD thesis, Yale University, 1984.

[20] A. Nicolau. Percolation Scheduling: A parallel compilation technique. Technical Report 85-678, Cornell University, 1984.

[21] A. Nicolau and J. Fisher. Measuring the parallelism available for Very Long Instruction Word architectures. *IEEE Transactions on Computers*, C-33:968–76, November 1984.

[22] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[23] M. Y. Wu and D. D. Gajski. A programming aid for hypercube architectures. *Accepted for publication in Journal of Supercomputing, to appear August 1988.*