ON THE COMBINATION OF HARDWARE AND SOFTWARE CONCURRENCY EXTRACTION METHODS

Augustus K. Uht[1]
University of California, San Diego
Dept. of Computer Science and Engineering, C-014
La Jolla, California 92093

Constantine D. Polychronopoulos[2]
University of Illinois at Urbana-Champaign
Center for Supercomputing Research and Development
Urbana, Illinois 61801

John F. Kolen[3]
University of California, San Diego
Dept. of Computer Science and Engineering, C-014
La Jolla, California 92093

Abstract

It has been shown that parallelism is a very promising alternative for enhancing computer performance. Parallelism, however, introduces much complexity to the programming effort. This has lead to the development of automatic concurrency extraction techniques. Prior work has demonstrated that static program restructuring via compiler based techniques provides a large degree of parallelism to the target machine. Purely hardware based extraction techniques (without software preprocessing) have also demonstrated significant (but lesser) degrees of parallelism. This paper considers the performance effects of the combination of both hardware and software techniques. The concurrency extracted from a given set of benchmarks by each technique separately, and together, is determined via simulations and/or analysis. The "common parallelism" extracted by the two methods is thus also considered, using new metrics. The analytic techniques for predicting the performance of specific programs are also described.

1. Introduction and Background

A variety of software and hardware techniques for improving computer performance have been proposed or implemented. Many of these schemes aim at extracting parallelism at different levels of granularity and at different phases of the program development and execution cycle. Parallelism can be extracted or specified at many different levels:

1. Task level
2. Routine or process level
3. Subroutine level
4. Loop level
5. Machine instruction level, or low-level [24]
6. Micro-level, e.g., micro-instruction overlap [12], pipelining

Parallelism may be *explicitly* specified by a user, or be *implicit* in the user's code. If implicit parallelism is to be exploited automatic *concurrency extraction* techniques must be used to detect the parallelism and schedule the resulting operations [14, 16, 13, 23, 24]. Automatic techniques are preferable for many practical reasons, including:

1. The programming task is made easier for the user. The underlying software/hardware system is transparent to the user.
2. Much pre-existing sequential code, e.g., "dusty decks," may be executed in parallel, without re-writing it.
3. Given the complexity of the parallelism specification and exploitation problem, automatic schemes are more effective on the average than manual optimizations.

In this paper we restrict ourselves to the consideration of automatic extraction methods applied at the loop and machine instruction levels (levels 4 and 5), although such techniques could, in principle, be applied to the other levels. In particular, modified level 5 methods could be used to execute micro-instructions concurrently. Although the techniques considered are applied at specific levels, they may also extract concurrency at and between other levels.

Traditionally, primarily static software-based methods have been applied to the loop level, while hardware-based methods have been applied to the machine-instruction level. More often than not, these two approaches have been considered separately, in exclusion of each other. There have also been questions raised about the parallelism exploited by each technique and the overlap thereof. These issues are considered in this paper.

A particular software technique, the Parafrase [8, 10] preprocessor is applied both separately, and in conjunction with, a hardware technique: the CONDEL-2 low-level concurrent machine model [21, 23], to the same set of benchmarks.

The major questions addressed by the experiments described below are:

1. What is the effect on performance (speedup) of applying both software and hardware concurrency extraction methods simultaneously to the same problem?
2. What parallelism is extracted by each method, and what is the *overlap*, or "common parallelism" of the two methods?
3. Can the speedups be predicted analytically, and if so, how?

[1]The first author was supported in part by the University of California at San Diego, and the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

[2]The second author was supported in part by the National Science Foundation under Grants No. NSF DCR84-10110 and NSF DCR84-06916, the U. S. Department of Energy under Grant No. DOE DE-FG02-85ER25001, and the IBM Donation.

[3]The third author is currently in the Department of Computer and Information Science, Ohio State University, Columbus, Ohio 43210. He was supported in part by the University of California at San Diego.

The first two questions lead to the following hypothesis:

> The total speedup of the combination of the two methods is greater than the methods taken separately, and is possibly greater than the sum of the two.

The former part of the hypothesis reflects the belief that the two methods will not hamper each other. Although one can contrive cases in which this is not true, if the software technique is aware of the limitations of the hardware technique, no reduction in performance should occur.

We arrive at the latter part of the hypothesis via the following reasoning. The compiler can extract much parallelism at the loop level which is (or can be made to be) over and above that which is extracted by the hardware, particularly in the case of DOALL[4] constructs with a large number of iterations. Likewise, the hardware may extract much low-level concurrency, particularly of a dynamic nature (i.e., from other types of loops and/or code with complex control flow), which software techniques may not be able to extract or exploit efficiently. In addition to these orthogonal performance contributors, the compiler can remove low-level concurrency inhibitors [21], resulting in synergistic performance improvement.

The hypothesis is tested in the experiments. New, generally applicable, metrics are developed and used to quantify and characterize the speedup overlap and the combined performance gain of the two techniques.

The remainder of this paper is as follows. The basic restricters of concurrency, dependencies, are briefly described in Section 2. The hardware functions and description are given in Section 3. The software functions and description are presented in Section 4. The combination of the two techniques is discussed in Section 5. The metrics used to determine and analyze performances are defined and described in Section 6. The performance estimation analysis methods are outlined in Section 7. The experimental results are given and discussed in Section 8. The conclusions of the paper are summarized in Section 9.

2. Dependencies

The necessary restrictions in typical code that prohibit many instructions from executing concurrently, and thus arise due to the constraints of the code itself, being independent of hardware restrictions, are called *program* [8] or *semantic* [21] *dependencies*.[5] There are two classes of dependencies, *data* and *procedural* (the latter is also called *control* or *branch*).

There are three types of data dependencies: flow, anti, and output dependencies [2]. A flow dependency is defined between two assignment statements, if a variable defined by the former statement is used in the latter statement. An antidependency is similarly defined, but in this case a variable used by the former statement is redefined in the latter statement. Output dependency is defined between two statements that write into the same variable (memory location). Theoretically, only flow dependencies need be enforced [3]. This ideal is closely approached by the two methods described in this paper.

Procedural dependencies arise due to the constraints imposed by the presence of branches in the instruction stream. The classic model assumes that all code after a dynamically occurring branch is dependent on the branch. This is not essential. Sets of specific dependencies may be defined to reduce these constraints. A minimal set of procedural dependencies is described in [21, 24]; this set is used by the hardware method described herein. Another reduced set, a bit more restrictive, is used by the software extraction method.

Both hardware and software concurrency extraction methods employ very reduced semantic dependency models.

3. Hardware Functions and Description

In this section we review general aspects of low-level concurrent machines, and describe the actual hardware model simulated.

The basic goal of hardware low-level concurrent machines is to automatically extract concurrency at the machine instruction level. By examining the instruction stream and detecting the semantic dependencies amongst the instructions, instructions whose effects are independent of each other may be executed concurrently, improving performance. Normally only a subset of the total instruction stream is examined by the hardware at a time, potentially restricting the parallelism extractable by the machine alone. The basic organization of these machines consists of an instruction scheduler and multiple functional units or Processing Elements (PE's). As the instruction stream is examined, independent instructions are detected and issued for concurrent execution.

Classic work done in the area of hardware low-level concurrency extraction is in [6, 17, 19, 20]. More recent work includes: [1, 5, 12, 22, 23, 25]. See [24] for a brief comparison of most of these techniques.

The hardware model used in our studies is an advanced low-level concurrent machine, CONDEL-2 [21, 23][6]; see Figure 3-1. This model achieves close to minimal data and branch dependencies. It also contains structures allowing some code sections to execute ahead of time, with no penalty if the results are not needed. Like other machines, the basic structure of CONDEL consists of a central instruction issuing unit simultaneously supplying multiple simple PE's[7] with instructions. The input to the processor is a typical single stream of machine instructions, consisting of simple assignment statements and branches. As is also typical, backward branches in the input stream are used to realize all looping constructs specified by the high-level language. No distinction is made between different types of loops (this is slightly modified for the purposes of two of the experiments; the modifications are described in Section 8). The static, i.e., lexicographic, instruction stream order is used; a subset of the total program is examined at a time (typically 16 or 32 static instructions), in a hardware *window* called the *Instruction Queue* (IQ) [25]. CONDEL appears to the user as a SISD machine, but internally it acts as an MIMD machine, with the instruction issuing unit performing the concurrency extraction, and hence the Single- to Multiple- instruction stream conversion.

Since the order of the code is independent of the dynamic control flow, at least within the window, it is possible to execute instruction *instances*[8] beyond branch *domains*[9]. This happens since both the code within the branch domain and the code after the branch domain are present simultaneously, regardless of the execution of the branch. The execution of code occurring after the branch domain is not dependent on the branch *per se*, and thus may execute concurrently with the branch. Therefore the ill effects of branches are reduced through *reduced procedural (branch) dependencies* [18, 21, 22, 24]. Descriptions of the specific necessary and sufficient reduced procedural dependencies may be found in [24].

Only flow dependencies are enforced for scalar assignments. Array (and pointer) assignment dependencies are also reduced, but to a lesser extent.

As each static instruction is loaded into the machine, the dependencies between the new instruction and those already loaded are computed. Computing the dependencies at load time, rather than at compile time, vastly reduces the memory bandwidth required for instruction fetches,[10] while keeping the degree of dependence amongst the instructions low. This occurs since the dependencies, several bits per instruction pair, do not have to be transmitted to the machine; also, the dependencies computed and realized by the

[4]DOALL loops contain independent iterations which may be executed completely in parallel.

[5]The terms *dependence* (see [19, 23]) and *dependency* (see [8]) are used by different researchers to mean slightly different things. For the purposes of this paper, they are considered to be the same.

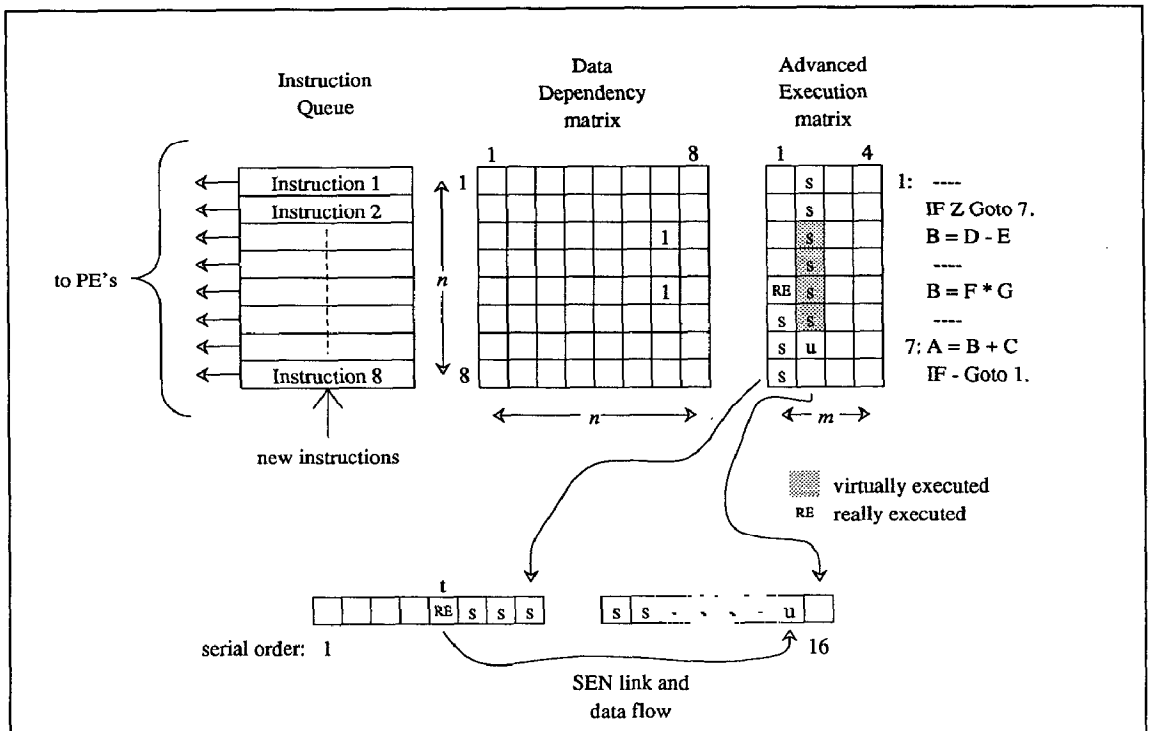
[6]Unless noted otherwise, all references to CONDEL in this paper imply this particular version.

[7]These elements have the complexity of semi-smart ALU's, possibly performing some assignment instruction decoding and operand fetching, as well as the usual execution.

[8]An instance of an instruction is the instruction executing in one iteration, whether it is within a loop or not.

[9]A branch domain consists of the code from the branch to its target.

[10]Alternatively, the instruction order could be specified explicitly to the machine, and eliminate the need to compute the dependencies in the hardware; however, this would eliminate the performance gains achieved by the hardware as is, gained via the scheduling of execution of instructions dynamically, as soon as possible.



Notes: the first branch is not taken in iteration 1, but is taken in iteration 2 (giving the virtual executions shown); two or more iterations are enabled. In the lower part of the drawing, the first two columns of the AE matrix are shown unfolded, and in their serial, or nominal execution, order.

Figure 3-1: Basic CONDEL-2 concurrency structures, with SEN example determination.

hardware are more dynamic than those computable by software. (Of course, software may want to compute its own set of dependencies for high-level concurrency detection.) Once an instruction is loaded, it (and other previously loaded instructions) may be executed. Over the course of code execution, dynamic state information is maintained, indicating which instances have been executed. During each execution cycle, an Executably Independent Calculator in the processor combines the dependency information with dynamic state information, to determine which instances may be executed in the current cycle. These instructions are issued in parallel, and the dynamic state is updated accordingly, whence the cycle repeats.

The dynamic execution state is held in two structures, the *Really Executed (RE)* and *Virtually Executed (VE)* bit matrices, each having n rows (one per Instruction Queue row or instruction), and m columns, one per instruction iteration. Typically, n is 16 to 32, and m is 4 to 8. An instruction instance is really executed when its specified operation has actually taken place, i.e., it is executed normally. Virtual execution occurs if the instance is disabled for execution by a branch executing true, i.e., in traditional terms, if a branch is taken and goes around the instance. An instruction instance can be virtually or really executed, but not both. In either case, instance execution results in the corresponding bit being set in the appropriate matrix. The OR of the RE and VE matrices gives the *Advanced Execution (AE)* matrix [25], shown in Figure 3-1.

The basic determinations of both instruction instance issuance for execution, and source linking (determining the inputs to the instance), are made by the *SEN* (Sink ENable) logic, a simplified version of which is:

$$SEN_t^u = RE_t \bullet DD_{row(t),row(u)} \bullet \prod_{s=t+1}^{u-1} (\overline{DD}_{row(s),row(u)} + VE_s)$$

(Refer to Figure 3-1 for the following discussion.) u is the serial index of the instruction iteration under consideration for execution, t is the serial index of an instruction iteration occurring serially prior to u, and s is the serial index of instruction iterations between t and u. DD is a binary Data Dependency matrix. A SEN_t^u is a 1 only if instance t is to supply its sink to the input of u, and u may execute, in the current cycle. This is only possible if t has really executed, i.e., a value exists for that instance's sink, and t is data dependent on u; also, all instances s between t and u that are data dependent with u must be virtually executed, otherwise they would be candidates to supply u with an input in the current or a later cycle. Only one SEN_t^u may be 1 for each instance u; if none are 1, the data does yet exist for u, and u may not be issued for execution. In the example of Figure 3-1, all of the conditions are met, so instance u gets its input B from the first iteration instance of instruction 5.

Using this as part of special hardware algorithms, data flow execution of the input code is achieved[11]. Performance is enhanced further via both the use of the reduced procedural dependency model [21, 22, 24], and the application of a simple form of branch prediction, allowing one or more instances of un-enabled loop iterations to execute ahead of time. Code execution is decoupled from memory updating (copying the results to memory), resulting in no time penalty upon a wrong guess.

Thus, many powerful techniques are used to enhance the low-level concurrency obtainable from nominally sequential code. The hardware is able to exploit more low-level concurrency than purely static techniques, since the hardware in effect generates and utilizes a dynamic dependency graph, which is normally less than or equally restrictive to a corresponding static graph. However, there may still be characteristics of the code which inhibit its concurrent execution. Additionally, with current hardware methods, only a portion of a program is considered at a time, restricting concurrency. It is also possible that more concurrency can be obtained from a program if more information is available about higher-level constructs. Therefore compiler-based methods should also be of use.

4. Compiler Functions and Description

Another concurrency extraction technique is program restructuring. In this approach parallelism is discovered and made explicit to the run-time environment by the compiler. Program restructuring is less dependent on the details of the target machine. Specific architectural

[11]This is for scalars, as well as many array accesses. See [24].

features can be exploited by a back-end compiler which is machine specific. Restructuring may be limited, however, in that it cannot take advantage of dynamic program characteristics only observable at run-time, e.g., dynamically computed loop-bounds and actual control flow. Also, restructuring techniques alone are not well-suited for exploiting low-level concurrency.

Parafrase applies to the input program a number of optimizations that are architecture dependent or independent. The first phase applies a number of transformations that are machine independent and which are always useful in aiding the vectorization/parallelization phase. Such optimizations include scalar renaming and expansion, subroutine expansion, dead code elimination, first order recurrence recognition, and many others. The second phase involves architecture specific optimizations. In this phase loops are recognized and translated to vector or parallel loops depending on whether the underlying architecture is a vector or multiprocessor machine.

The Parafrase restructurer is able to discover parallelism at several different granularity levels. Although parallelism detection at the statement and operation level is possible through Parafrase, this capability was not used in analyzing our benchmarks, since low level concurrency extraction appears to be done more effectively with hardware methods. In this paper Parafrase was used to restructure the benchmarks and discover parallelism at the loop level. Assuming a multiprocessor machine with CONDEL processors, parallelism exploitation at lower levels (e.g., within each loop iteration) is left to the hardware.

Automatic program restructuring from a serial to a parallel form is achieved through data and control dependency analysis by the compiler. Data and control dependencies define a partial order on the statements of a program. The Parafrase compiler enforces a close to minimal set of data dependencies. During actual execution, this order must be obeyed in order to guarantee correctness of the results. The program dependency graph is constructed by the compiler such that nodes correspond to statements in the source program and arcs represent data and control dependencies between statements.

Based on the data dependency graph, Parafrase [11] applies a number of transformations that restructure the program into a parallel form. Parallel constructs are explicitly specified in the output code. The most important parallel constructs (that are relevant to this work) are several types of parallel and recurrence loops. In particular, we have the following types of loops in a typical Parafrase output. DOSERIAL loops are purely serial loops. In a DOSERIAL loop all statements are involved in a data dependency cycle. DOALLs are fully parallel loops; all the iterations of a DOALL can execute simultaneously. A more general type of parallel loop is the DOACROSS loop [4, 15]. A DOACROSS loop contains a dependency cycle (or a backward dependency) that involves only some of the statements in the loop body. Successive iterations of DOACROSS loops can be partially overlapped. DOSERIAL and DOALL loops can be thought of as special cases of DOACROSS loops when the dependency cycle involves all statements in the loop or if the cycle involves no statements, respectively. Certain types of DOWHILE or EXIT-IF loops are treated as DOALL's or DOACROSS's by Parafrase, depending on the loop dependency graph.

Thus, software methods are able to detect much parallelism at the loop level.

5. Combining Compiler and Hardware Solutions

The basic scheme for combining the hardware and software concurrency extraction methods is straightforward. Code to be executed is first passed through the program restructuring compiler, and then to one or more of the concurrent machines for actual execution. The eventual goal for the compiler is to

- perform an accurate dependency analysis,
- remove low-level concurrency inhibitors from the code, and
- detect parallelism at a high-level, i.e., determine which loops may be executed in a DOALL fashion.

The goal of the concurrent machine is to execute the restructured code as concurrently and efficiently as possible. In the case of DOALL loops, multiple processors may be used to maximize the performance, each machine executing a subset of the total number of iterations.

6. Metrics

6.1. Definitions

It is assumed throughout that each instruction or fundamental operation takes one cycle to execute.

Definition 1: T_1 is the time to execute a benchmark strictly sequentially.

Definition 2: $S_* \equiv \frac{T_1}{T_{par}}$. Speedups are computed simply by dividing the time to execute the program sequentially by the time to execute the program in some parallel fashion (*).

Definition 3: S_T is the total actual speedup obtained with multiple methods combined, e.g., in the case of this work, the output of the Parafrase compiler is used as the input to the CONDEL-2 hardware model, via the method given in Section 8.4.4.

Definition 4: ω is a measure of the *overlap* in concurrency, or "common parallelism", extracted by the set of methods used.

$$\omega = \frac{S_{nom\text{-}pk} - S_T}{S_{nom\text{-}pk} - \max[S_i]}.$$

where: S_i is the speedup due to extraction method i alone, and

$$S_{nom\text{-}pk} = \prod_{\text{all } i} S_i.$$

In this work: $S_i \in \{S_{Hardware}, S_{Software}\}$. If ω is between zero and one, it indicates the degree of overlap between the methods. When it is greater than one, a degenerative situation exists. If equal to one, there is complete overlap, and no gain is obtained from the combination of the methods. When equal to zero, there is no overlap, and the parallelism obtained from each technique is apparently fully utilized. ω is undefined if any $S_i \leq 1$; (it is hard to quantify an overlap with nothing or something negative, which is what such speedups imply). $S_{nom\text{-}pk}$ is the nominal-peak speedup one might intuitively expect when completely orthogonal performance enhancement methods are cascaded. It is not the "maximum" speedup obtainable.

Definition 5: σ is the *synergy* indicator, or measure of effectiveness of the combination of the different concurrency extraction techniques.

$$\sigma \equiv \frac{S_T}{S_{nom\text{-}pk}}$$

When σ is greater than one, synergy exists, in that the speedup obtained by the combination is greater than the product of the individual speedups. If σ is less than or equal to one, and greater than zero, the indication is that no synergy exists, although combining the methods may still produce a performance gain; in this case, the metric indicates how much of the nominal- peak performance is obtained.

Definition 6: S_v gives the effective speedup of the combination of the methods over the methods alone; formally:

$$S_v \equiv \frac{S_T}{\max[S_i]}$$

If S_v is greater than one, the combination of the methods has a positive effect on performance.

6.2. Example and Discussion

An example of the use of the metrics is shown in Figure 6-1. For a

S_T	ω	σ	S_v
100	-1.25	2.0	10.0
75	-0.63	1.5	7.5
S_{nom-pk}= 50	0.0	1.0	5.0
40	0.25	0.80	4.0
30	0.50	0.60	3.0
20	0.75	0.40	2.0
$S_{Software}$= 10	1.0	0.20	1.0
7	1.08	0.14	0.7
$S_{Hardware}$= 5	1.125	0.10	0.5
3	1.18	0.06	0.3
0	1.25	0.0	0.0

Note: The metrics are computed for various values of S_T, given: $S_{Hardware}=5.0$ and $S_{Software}=10.0$.

Table 6-1: Example illustrating the behavior of the metrics.

single set of $S_{Software}$ and $S_{Hardware}$ values, different combined performance figures S_T are posited, with the corresponding metric values, showing the behavior of the metrics in different cases.

The overlap ω is limited in that it does not indicate in what way performance methods overlap; it only gives a rough outside indication of the common parallelism. To completely understand the dynamics, it is still necessary to examine the data in detail; in the case of this paper, this means to examine the methods and their executions of the benchmarks in depth. The synergy indicator σ presupposes that S_{nom-pk} is a reasonable likely maximum; not everyone may agree with this. The combination speedup indicator S_v is useful in describing what the performance gains of the combination are over and above what was the best achieved with the enhancement methods applied separately.[12] When used together, these three metrics provide a good characterization of the effects of combining performance enhancement techniques.

7. Performance Estimation

It is desirable to be able to predict performance of any machine or machine model. Although analyzing arbitrary code to estimate its performance on a particular machine may be prohibitively unwieldy, code with relatively simple control flow may be more amenable to reasonable analysis. Such is the case with CONDEL, particularly when executing loops. Simple loops consisting solely of assignment statements and a loop-forming backward branch are considered first, followed by consideration of similar loops containing IF-THEN's.

Without IF-THEN's, un-nested loops may be executed sequentially in time $T_1=K+(L\times r)+M$, where L is the static length of the loop, r is the number of iterations of the loop to be executed, and K and M are the lengths of the code segments before and after the loop, resp. As determined in [7], the loop may be executed concurrently on CONDEL in time dependent on the length of the *dependency* [9] or *computation* [21] *cycle* X. Since CONDEL does not currently have the ability to generate multiple values of the same index at the same time (as would be useful for executing DOALL's), $X\geq 1$. CONDEL's maximum performance occurs if it is able to execute one instance of every instruction within a loop every cycle (the instances may be in different iterations), i.e., with X equal to 1; this is called *saturation*. X is determined by finding the longest cyclic path between successive iterations of static instructions within the loop. Code segments K and M are executed concurrently with the loop, when dependencies allow.

The resulting concurrent execution time is then: $T_{par}=W+(X\times r)+Y$, where W and Y are the longest thread lengths of the sections of code segments extending before (resp. after) the longest cycle, and dependent on the cycle. These segments contain K and M and may be overlapped with the loop. As r becomes large, the resulting speed-up is: L/X, which is the same result as that obtained for concurrent execution of DOACROSS loops in [4, 15]. It is also the same as the basic limit on pipeline performance.

When IF-THEN's are present, the length of a dependency cycle may vary as the code is executed, depending on the data values. An example of this is a recurrence, i.e., the value of an IF conditional may be computed within the same THEN in a prior iteration. Also, the effective value of L changes as IF's execute true in some iterations (decreasing the effective loop length), and false in others. The dependency cycle variation affects the concurrent execution time, the loop length variation affects the sequential time. In such loops, upper and lower bounds on the expected performance are determined by finding X and L for the appropriate combinations of branches taken and not taken, computing the possible speedup values, and using the extremes as the bounds on performance.

Thus, CONDEL executes code (in fact, all loops) in a concurrent DOACROSS fashion, and also possibly with dynamically changing X. Consideration of predicted performances with actual values is given in Section 8.4.1. Other code and machine situations are considered in [7].

Execution times of other constructs on ideal machines are easily determined. A DOALL loop will execute in the time necessary to execute a single iteration either concurrently or sequentially, depending on the machine, if synchronization is neglected. If a reasonable worst case for synchronization is assumed, i.e., each iteration is skewed from its predecessor by one cycle (linear overhead), then the DOALL time is the same as a DOACROSS time for the same loop with $X=1$. Detected and executable recurrences take logarithmic time to execute in the ideal case. In other cases, DOACROSS loops with $X=1$ are again assumed.

8. Experiments

8.1. Introduction and Outline

Experiments were performed to determine the effectiveness of each concurrency extraction method separately and together. The same set of benchmarks is used throughout the experiments. The execution of the benchmarks is simulated or determined analytically for several possible system cases:

1. Hardware method of low-level concurrency extraction used alone.
2. Software restructuring used alone, applied to a multiprocessor model.
3. The combination of the two methods, assuming the hardware has no special capabilities to execute DOALL loops or recurrences.
4. The combination of the two methods, assuming the hardware is capable of executing DOALL and recurrence constructs in as parallel a fashion as possible.

In the first case, the CONDEL low-level concurrent machine model is used alone. For the last three cases, the Parafrase restructuring compiler developed at the University of Illinois is used to generate the restructured code.

The specific methodologies and results for each case are described below, after the methods for using the restructured code are described, and the benchmarks are presented.

8.2. Using the Restructured Code

The Parafrase output was modified to a form that would occur if the compiler had been designed to produce code for CONDEL machines. The major compile-time optimizations that were assumed were: loop parallelization, dead-code elimination, common subexpression

[12] In fact, one can imagine the usefulness of computing a series of S_v, one for each i, defined as:

$$S_v^{\,i}\equiv\frac{S_T}{S_i}.$$

Each value indicates the benefit of the combination relative to one particular method, not just the best. Looking at groups of these figures for a variety of benchmarks would indicate which method provides the larger improvement on average. We do not use this modification here.

elimination, in-line subroutine expansion, array renaming to satisfy the single assignment rule,[13] and dependency cycle reduction.

As previously mentioned, the basic strategy is to pass the benchmarks through Parafrase, and then through the CONDEL simulator (or determine the performance using the analytical techniques previously mentioned). Since no software exists to take Parafrase output and convert it to CONDEL assembly code, the following revised approach is used:

1. The multiprocessor or MES (Multiple Execution Scalar [10]) output of the Parafrase compiler (in modified FORTRAN) is used as the starting point. This code consists of a combination of the following components:

   DOALL, DOACROSS, and DOSERIAL loops, and scalar code. (DOWHILE loops must be handled specially, as Parafrase does not accept them directly. There were none in the benchmarks studied.) The Parafrase output is modified to a form that would occur if the compiler had been designed to produce code for CONDEL machines.

2. The different components of the Parafrase output are hand-coded into CONDEL Assembly code, with the different components treated as follows:

   - DOALL loops and recurrence calculations: this depends on the particular system model being considered. If no synchronization overhead is assumed, then the execution time of the constructs is primarily determined by the execution time of a single iteration on a CONDEL machine, concurrently or sequentially, as appropriate. In the case of recurrences, a penalty equal to the logarithm of the number of iterations is added to the single iteration time. If either significant overhead is assumed, or only a single concurrent machine is available to execute the code, then the constructs are treated as DOACROSS loops with delay (or offset) of one cycle.
   - Other code, e.g., DOACROSS, DOSERIAL, and DOWHILE loops, and scalar code: coded and simulated or analyzed directly, assuming a single CONDEL processor (with or without multiple PE's, depending on the system in question).

3. The final execution time of the combined system is computed from the combination of the two times above, and the speedups are computed normally.

8.3. Benchmarks

The benchmarks used consist of selected routines from the Whetstone and LINPACK-BLAS program suites. The programs were selected to obtain a variety of code characteristics in the test set. The benchmarks are:

1. Whetstone modules:
   a. Module 1 (Whet-1): Simple identifiers (assignments to scalars).
   b. Module 2 (Whet-2): Array elements (assignments to arrays).
   c. Module 4 (Whet-4): Conditional jumps.
   d. Module 8 (Whet-8): Procedure calls.
2. LINPACK-BLAS routines:
   a. ISAMAX: Find the index of the element of a vector with the maximum absolute value.
   b. SASUM: Take the sum of absolute values of a vector.
   c. SAXPY: Constant times a vector plus a vector.
   d. SDOT: Form the dot product of two vectors.
   e. SROT: Apply a plane rotation.

All of the benchmarks consisted of a single loop. In most cases, the loop was executed for 32 iterations. Whet-1, Whet-2, and Whet-4 were executed for more iterations, but the speedups would not have changed significantly if 32 iterations had been assumed.

8.4. Concurrency System Method Descriptions

For the baseline cases (used to generate T_1), the high level language representations of the benchmarks were hand-coded directly into assembly language without assuming any significant compiler assists, then assembled, and executed on the `simcd` simulator [21], assuming a sequential version of CONDEL-2 (this is achieved by setting the Instruction Queue length to 1 and turning off subroutine expansion).

The same values for T_1 were used for all of the different models considered. In this way the compiler effects are kept clear. (The Parafrase output could also be used as the input to a sequential machine, and in many cases would improve the sequential machine performance as well. With one exception [Whet-8], the results of our experiments would not change substantially if a T_1 figure generated thus were used.[14])

8.4.1. Hardware-Only Experiments

The same codes used to generate the T_1 numbers (for the sequential or baseline case) were used in these experiments as input to the same simulator assuming the CONDEL-2 concurrent machine model (previously described in Section 3). The results are presented in Table 8-1 in the S_H column.

The analytical performance estimation techniques of Section 7 were applied to these simulations and some of those of Section 8.4.3 and were found to accurately predict performance. The actual values were either practically equal to the predicted values, off by one cycle at most, or were within the range of values predicted. Therefore these methods were used in the remainder of the experiments to produce performance estimates as needed, normally in conjunction with simulations.

8.4.2. Compiler-Only Experiments

In this experiment the restructuring technique was used alone; no low-level concurrent machine was assumed. It was assumed that the restructured code, generated as described above, was executed on an MES architecture. This is a multiprocessor system composed of sequential scalar processors. In order to directly compare results with the other systems, a CONDEL instruction set architecture was assumed for each of the scalar processors, executing the processor input code sequentially. As is normal with the MES model, DOALL, DOACROSS, and recurrence code sections were spread amongst the scalar processors, typically on an iteration per processor basis. No time penalty was assumed to spawn multiple iterations simultaneously, or to synchronize the iterations. No more than 32 processors are used at a time. The speedup results for the benchmarks are computed from serial and parallel execution times (T_1 and T_{par}), and are shown in Table 8-1, in the S_{S-MES} column. Note that in some cases the speedups are greater than 32; this is due to Parafrase's elimination of some of the loop overhead when executing DOALL loops.

8.4.3. Combined Methods (Restricted) Experiments

In this system the restructured code was applied to a single unmodified CONDEL model executing the code in its normal concurrent fashion. DOALL loops and recurrence calculations were executed, in effect, as DOACROSS loops. Normally, CONDEL executes all loops other than DOALL's and recurrences as DOACROSS loops with the added benefits of low-level concurrency

[13] Array renaming, as applied to many CONDEL code cases, assigns multiple distinct pointers to some of the different array accessing instructions that access the same array, making such accesses data independent. In such cases, no new array storage was used. In effect, single assignment restrictions are realized with the usual benefits ensuing, but without the usual overhead of extra memory usage.

[14] In the case of Whet-8, a subroutine expansion by Parafrase eliminates a lot of overhead normally occurring in the sequential case; the speedups would be reduced in this case by about 44% if the improved sequential number were used.

Benchmark	Code class	S_H (*4)	S_{SEC} (*4)	S_{S-MES} (*5)	S_T (*5)	S_{nom-pk}	ω	σ	S_v
Whet-1	1	1.61	2.41	1.25	2.41	2.01	-1.00	1.20	1.50
Whet-2	1	1.80	2.42	1.20	2.42	2.16	-0.72	1.12	1.34
Whet-4	2A	1.45	1.52	1.19	1.52	2.20	0.91	0.69	1.05
Whet-8	4	11.64*	11.91*	12.19	42.70	141.89	0.76	0.30	3.50
ISAMAX	2B	1.92→ 4.51 (3.54)	5.45→ 7.17 (5.89)***	1.83→ 2.13	5.45→ 7.17	3.51→ 9.61	-1.22→ 0.48	1.55→ 0.75	2.84→ 1.59
SASUM	3	8.80→ 9.60 (9.33)*	8.80→ 9.60 (9.33)*,**	18.0→ 18.67	25.71→ 28.00	158.4→ 179.2	0.95→ 0.94	0.16→ 0.16	1.43→ 1.50
SAXPY	4	2.60	6.95*	36.70	51.4	95.4	0.75	0.54	1.40
SDOT	3	6.14*	6.14*,**	12.60	17.46	77.36	0.92	0.23	1.39
SROT	4	3.70	13.00*	43.73	96.20	161.80	0.56	0.59	2.20

Notes: * These benchmarks executed in *saturation*: one instance (instruction iteration) of each static instruction in the loop executes per cycle.

** No change from the S_H value, indicating that maximum speedup was achieved with the base hardware alone.

*** Saturation is not achieved in this case due to the nature of the algorithm; a dependency cycle with variable length greater than one (actually from 2 to 3) exists between a branch test and an instruction within the branch's domain.

*4 With the exception of Whet-2, CONDEL-2 had 32 or fewer PE's. In many cases, including this one, equivalent speedups should be obtainable with far less hardware [21, 22].

*5 32 processors were assumed for both models. The performance results were not constrained by this number.

A range of numbers indicates a control dependency on run-time data; the extremes are shown.

Numbers in parenthesis are the results of simulations on single sets of random data.

The values of S_{nom-pk}, ω, σ and S_v were derived from the S_H, S_{S-MES} and S_T results.

Table 8-1: Performance results and comparisons of the concurrency extraction techniques, separately and together.

extraction (assuming the loops fit into the Instruction Queue). This system is called the SEC model, for Single Execution Concurrent system. This model illustrates the effect of using sophisticated compilation techniques in conjunction with a low-level concurrent machine alone. It is indicative of a combined system with large DOALL and recurrence synchronization overhead. The speedups obtained from a combination of simulations and estimates are shown in Table 8-1 in the S_{SEC} column.

8.4.4. Combined Methods (Unrestricted) Experiments

This model also uses the combined software and hardware methods, but to a maximum extent. It is assumed that the equivalent of multiple CONDEL processors exist to execute DOALL and recurrence calculations in as parallel a fashion as possible (a similar assumption for simple DOALL's and recurrences is to have a single processor with the capability of generating multiple indices simultaneously; see Section 8.5). In fact, the low-level concurrency within a DOALL iteration is determined by estimating the execution time of one loop iteration on CONDEL in a concurrent fashion.

This model is called the MEC model, for Multiple Execution Concurrent machine model. The speedup results of this model, shown in the S_T column in Table 8-1, indicate an upper bound on the performance achievable by the combination of the hardware and software concurrency extraction techniques as currently formulated.

The synchronization overhead of DOALL's and recurrence calculations is assumed to be zero. This is not necessarily the case, but the variation in such overhead amongst typical machines is such that it is impossible to estimate it here. A range of possible performance is indicated by considering both the SEC and MEC results.

8.5. Analysis and Discussion of the Results

The results of the three sets of experiments are shown in Table 8-1. This section begins with general discussions of the separate hardware and software methods (S_H and S_{S-MES}), followed by specific detailed analysis of the individual benchmarks' results occurring from the performance enhancement methods both separately and together. The section concludes with general comments on the combination of the two concurrency extraction techniques.

8.5.1. General Discussion of the Unmodified Hardware-Based Methods

The hardware speedup S_H and combination (without DOALL, etc.) speedup S_{SEC} are now analyzed further. The figures in the S_H column demonstrate that CONDEL alone is able to achieve respectable performance gains. With software preprocessing (see the S_{SEC} column), the results are uniformly better or equal. Many of the benchmark loops executed in saturation (see the "*" note in the table). Code executing in saturation on CONDEL is a maximum performance situation since loop counters can only be incremented once per cycle, resulting in a kind of DOACROSS execution of DOALL loops. Although it is true that the instruction set of CONDEL could conceivably be modified to generate multiple loop indices at once, it is not clear that this is the most efficient thing to do, due to CONDEL hardware cost constraints, and when the relative overheads of the hardware and software based methods are considered.

Since one instance of each static instruction executes per cycle, the limit on speedup is equal to the size (length) of the code within the loop; thus, loops executing in saturation have speedups close in value to their loop sizes. This indicates that loop unrolling could be used to increase the loop size and hence the speedup. This is limited, however, as the cost of the machine is very dependent on the size of the window, and the loop must fit in the window for saturation to

occur.[15]

8.5.2. General Discussion of the Software-Based Method

We observe that automatic program restructuring did not result in significant parallelism improvement in the case of most of the Whetstone routines (except for Module 8). The explanation of the failure of Parafrase in this case lies with the structure of these routines. The first three modules are simple loops that are mainly serial. In most cases parallelism is present at the operation or statement level. Even though Parafrase is quite effective in discovering loop parallelism (as in Module 8), we observe that parallelism exploitation through CONDEL was superior.

In the case of the BLAS routines, however, Parafrase performed clearly better than CONDEL (except for the ISAMAX routine, which contained dynamic control flow dependent on the run-time data). This is again due to the structure of the BLAS routines. These routines carry out elementary vector operations. Parafrase is very powerful in discovering parallel and vector loops. In addition Parafrase can recognize first degree recurrences and can substitute them with an equivalent parallel algorithm automatically. However, CONDEL is less able to recognize parallelism at the loop level. A necessary part of this process is the analysis of array subscripts. Performing array subscript analysis by the hardware is more complicated and more costly. A compiler however can perform the same task relatively easily. The superiority of Parafrase over CONDEL for such code is liable to be greater as the number of loop iterations to be executed increases, assuming most of the computation is, or can be made to be, in DOALL constructs and the scheduling overhead can be kept down.

8.5.3. Comments on Specific Benchmarks

In this section interesting aspects of specific benchmarks and their execution on the various models are discussed. The benchmarks are considered by common characteristics and are thus grouped in code Classes (see Table 8-1).

Class 1. These are basically sequential programs, in which X is large in both cases. Nonetheless, it was possible to reduce these dependency cycle lengths via intelligent compilation (estimated) of assignment statements. This is achieved by carefully choosing the order of evaluation of the long assignment statements in both of the programs, such that a High Level Language statement input which is in the critical path (longest dependency cycle) is moved as close as possible (in terms of instructions to be executed) to the machine instruction level statement output. This does little for the MES (software only) model, but greatly aids the MEC (combined) model.

Class 2. These two benchmarks originally contained relatively complex control flows, which were simplified by the software.

Class 2A. The Whet-4 code originally contained 9 partially overlapped (unstructured) branches within its loop. CONDEL was able to execute some of the branches concurrently, but was able to execute the code even faster after Parafrase had converted the control flow to a structured form. The new code contained 6 disjoint branches (no inter-branch dependencies). Although the improvement in performance of the combination of the methods was the smallest of all of the benchmarks ($S_v = 1.05$), the enhanced control flow could well be more significant when executing other programs concurrently.

Class 2B. The ISAMAX loop contains a forward branch that varies the dependency cycle length. In the original version of the code, there was also a procedure call within the branch's domain, and the call could not be expanded at run-time. Therefore the execution of the code on CONDEL was severely hampered when the branch was not taken, causing the call to be executed and resulting in the flushing of the Instruction Queue (equivalent to flushing a pipeline). As a result of the common subexpression elimination of the compiler, the call was eliminated, the Queue did not need to be flushed, and the dependency cycle length was reduced.

Referring to the S_T figures, it is also of note that the best absolute performance (71 cycles) gave the lower speedup (5.45); this was for the best performing control flow. This is because the relative change in performance due to different control flow assumptions was greater for the sequential case (the two T_1 values differing by a factor of 1.91), than the concurrent case (the two T_{par} values differing by a factor of 1.45).

The two resulting effects are an improvement in the performance and a reduction in the range of the cycle length, and hence speedups.

Class 3. The overlap ω occurring in each of these two benchmarks is large (greater than 0.90), and is confirmed by inspection, in that both the software and hardware techniques are getting most of their parallelism at the loop level. However, the software method does better because it is able to transform each code into a DOALL loop followed by a recurrence calculation, whereas the hardware method alone must execute the loop in a DOACROSS fashion.

Class 4. The characteristics of these results are similar to those of Class 3, but with the following exceptions. Parafrase is able to transform each program into a single DOALL (no recurrences). Therefore the MES model does very well by itself. The contribution of low-level concurrency reduction by the hardware improves matters even more so in the combined model. This is particularly true with Whet-8, in which there is a large component of serial code to begin with, but whose negative effect is reduced dramatically by the use of CONDEL. This is a particularly good instance of dealing effectively with the Amdahl effect. These characteristics are reflected in the comparatively low overlaps of all three of these benchmarks. For both SAXPY and SROT, compiler renaming via subscript analysis allows the loops to be executed quickly on the straight CONDEL model (SEC) since unnecessary dependency cycles are eliminated.

8.5.4. General Comments

Comparing the results of the two combined method models, SEC and MEC (S_T column), it is clear that for Class 1 and 2 type codes there is little benefit in having DOALL-like constructs. Conversely, if control flow is simple and loop iterations are independent, the constructs are of great use, as is shown by the Class 3 and 4 code results. The difference between the results of the two models also indicates the potential negative performance effects of synchronization overhead.

Although the overlap between the hardware and software concurrency extraction methods is often high (five or six out of the nine benchmarks), the effect of intelligently combining both the hardware and software concurrency extraction techniques is significantly positive, as evidenced by the generally good S_v (combination gain) figures. In a few cases (Whet-1, Whet-2, and one part of ISAMAX) synergy did in fact occur as we have defined it.

9. Summary and Conclusions

In this paper, specific hardware and software based concurrency extraction methods were described and applied both separately and together to the execution of a set of benchmark programs. Both techniques extracted concurrency in varying degrees from three levels: subroutine, loop, and machine instruction. The common parallelism extracted by the two methods was often high. However, it was sufficiently different (in some cases synergy occurred) that the combination of the two techniques, including the architecture

[15]The converse case, of a loop not fitting into the window, can be handled in many cases by loop fision.
All of the loops of these experiments fit into the window, which for all but the Whet-2 benchmark simulations was set to length $n = 32$ or less.

directed compilation, produced gains significantly greater than the methods used separately.

Compiler optimizations and program transformations are necessary to achieve the best results. The compiler has the ability to perform optimizations not only within a particular subroutine, but also across subroutine boundaries. Hardware routines are also able to do this, but not in all cases, and potentially less efficiently than software schemes. Also the compiler has global information about the entire program while the hardware can only handle a rather small portion of the code at any given time.

On the other hand the compiler is sometimes forced to make conservative assumptions about data dependencies when not enough information is present at compiler time. For example, superfluous dependencies may be assumed by the compiler when loop bounds or the dynamic control flow are not known. Since the hardware checks dependencies at run-time, it has all the necessary information to detect only true dependencies, or close to them. Thus, hardware based methods are also necessary for the best performance.

Our experiments, although limited in scope, support the original hypothesis, i.e., that combining program restructuring with clever hardware design should yield better performance than using each scheme separately.

Also, the new overlap, synergy and combination gain metrics proposed are useful in characterizing the effects of combining multiple performance enhancement methods.

References

[1] Acosta, R. D., Kjelstrup, J., and Torng, H. C. An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors. *IEEE Transactions on Computers* C-35:815-828, September, 1986.

[2] Banerjee, U. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October, 1979. Available as DCS Report No. UIUCDCS-R-79-989.

[3] Chamberlin, D. D. The Single-Assignment Approach to Parallel Processing. In *Fall Joint Computer Conference*, pages 263-269. AFIPS, 1971.

[4] Cytron, R. G. Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract). In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836-844. Pennsylvania State University and the IEEE Computer Society, August, 1986.

[5] Hwu, W. and Patt, Y. HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 297-306. ACM-IEEE, June, 1986.

[6] Keller, R. M. Look-Ahead Processors. *ACM Computing Surveys* 7(4):177-195, December, 1975.

[7] Kolen, J. F. Characterization of Concurrently Executed Programs. 1987. Undergraduate project report, Dept. of Electrical Engineering and Computer Sciences, University of California at San Diego, La Jolla, CA.

[8] Kuck, D. J., Muraoka, Y. and Chen, S.-C. On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup. *IEEE Transactions on Computers* C-21(12):1293-1310, December, 1972.

[9] Kuck, D. J. A Survey of Parallel Machine Organization and Programming. *ACM Computing Surveys* 9(1):29-59, March, 1977.

[10] Kuck, D. J. *The Structure of Computers and Computations.* John Wiley & Sons, New York, NY, 1978.

[11] Kuck, D. J., Kuhn, R. H., Leasure, B., and Wolfe, M. The Structure of an Advanced Vectorizer for Pipelined Processors. In *Proceedings of the Fourth International Computer Software and Applications Conference.* ACM, October, 1980.

[12] Patt, Y., Hwu, W., and Shebanow, M. HPS, a New Microarchitecture: Rationale and Introduction. In *Proceedings of MICRO-18*, pages 103-108. ACM, December, 1985.

[13] Polychronopoulos, C. D., Kuck, D. J., and Padua, D. A. Utilizing Multidimensional Loop Parallelism on Large-Scale Parallel Processor Systems. *IEEE Transactions on Computers* , publication date unknown. Accepted for publication as of September 1987.

[14] Polychronopoulos, C. D. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, University of Illinois at Urbana-Champaign, August, 1986. Available as Center for Supercomputing Research and Development Tech. Report CSRD No. 595.

[15] Polychronopoulos, C. D. and Banerjee, U. Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds. *IEEE Transactions on Computers* , April, 1987. Special Issue on Parallel and Distributed Processing.

[16] Polychronopoulos, C. D. and Kuck, D. J. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers* , December, 1987. Special Issue on Supercomputing.

[17] Thorton, J. E. Parallel Operation in the Control Data 6600. In *Proceedings of the Fall Joint Computer Conference*, pages 33-40. AFIPS, 1964.

[18] Tjaden, G. S. *Representation and Detection of Concurrency Using Ordering Matrices.* PhD thesis, The Johns Hopkins University, 1972.

[19] Tjaden, G. S. and Flynn, M. J. Representation of Concurrency with Ordering Matrices. *IEEE Transactions on Computers* C-22(8):752-761, August, 1973.

[20] Tomasulo, R. M. An Efficient Algorithm for Expoiting Multiple Arithmetic Units. *IBM Journal* :25-33, January, 1967.

[21] Uht, A. K. *Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, December, 1985. Available from University Microfilms International, Ann Arbor, Michigan, U.S.A.

[22] Uht, A. K. An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences.* University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, January, 1986.

[23] Uht, A. K. and Wedig, R. G. Hardware Extraction of Low-level Concurrency from Serial Instruction Streams. In *Proceedings of the International Conference on Parallel Processing*, pages 729-736. IEEE Computer Society and the Association for Computing Machinery, August, 1986.

[24] Uht, A. K. Incremental Performance Contributions of Hardware Concurrency Extraction Techniques. In *Proceedings of the International Conference on Supercomputing, Athens, Greece.* Computer Technology Institute, Greece, in cooperation with the Association for Computing Machinery, IFIP, et al, June, 1987. Springer-Verlag Lecture Note Series. In publication.

[25] Wedig, R. G. *Detection of Concurrency in Directly Executed Language Instruction Streams.* PhD thesis, Stanford University, June, 1982.