In: Languages and Compilers for Parallel Computing, Research Monographs in Parallel and Distributed Computing. The MIT Press, 1990. From: 1989 Workshop on Programming Languages and Compilers for Parallel Computing.

Desirable Code Transformations for a Concurrent Machine

Augustus K. Uht
Dept. of Computer Science and Engineering, C-014
University of California, San Diego
La Jolla, California 92093
(619) 534-6181
uht@cs.ucsd.edu

Abstract[1]
It has been demonstrated that low-level concurrent machines may extract a significant amount of concurrency from code. This is particularly true when compiler-based parallelization enhancements are used in conjunction with the hardware. However, this has been demonstrated without tailoring a compiler specifically to the machine. Machine-specific and general optimizations applied to the code at compile time may have a large effect on the total system performance obtainable. In this paper, such transformations are developed or identified, described, and evaluated. Significant performance gains resulting from the transformations are demonstrated.

1. Introduction

It is desirable to execute code as concurrently as possible, and hence with maximum performance. There are roughly two approaches to executing code concurrently: *explicit* and *implicit* parallelization. Explicit techniques force the programmer to schedule the parallel execution of his code by hand, a tedious process at best; however, this method has the benefit of effectively utilizing user knowledge of the problem/code. Implicit parallelization allows a certain amount of ignorance on the part of the user, and is also able to be used on existing codes ("dusty decks"). Our work is based on the implicit approach.

---

[1]This work was supported by the Semiconductor Research Corporation under Agreement No. 82-11-007, by Carnegie-Mellon University, and by the University of California, San Diego.

This paper is partially based on work originally developed in [15]. Part of this work may appear in a journal article currently undergoing review [21].

Employing either hardware [1, 11, 14, 17, 18] or software [2, 4, 5, 7] methods alone tends to restrict the concurrency obtainable, as well as requiring a very complex system, be it software or hardware. Also, the compile times of the software methods can be large [6] relative to the size of the code being compiled; for programs that are compiled many times and executed only once or twice, this is a severe disadvantage. In most of the hardware-only techniques, the speedups obtained are significant but not great.

In [19] the CONDEL-2 hardware concurrent model [17, 18] is used in conjunction with a software-based concurrency extraction method, the Parafrase compiler [8], demonstrating very significant speedups of the joint method. In this work the *modus operandi* was one of discovery: various concurrency enhancing optimizations of Parafrase were blindly applied to code, which was then executed by the CONDEL-2 model. In this paper we seek to refine the approach, to specifically tailor software-based code optimizations or transformations to the CONDEL-2 model.[2] Although the transformations are developed for a particular machine model, most of them are applicable to both static instruction stream (see Section 2.1) and concurrent machines in general.

Therefore, it is the goal of this paper to present the performance effects of an integrated compiler/concurrent computer system. In particular, the CONDEL-2 model is described in Section 2, analyzing the model and determining beneficial characteristics that should be demonstrated by the input code. In Section 3 these characteristics are developed into a set of low-overhead code transformations that can be applied to the input code, either as pre-processing of the high-level code, or as machine-specific optimizations in the back end of a compiler. The experimental system is described in Section 4. The effects of the application of the code transformations on the benchmarks are given in Section 5, with the overall performance results presented and analyzed in Section 6. Our conclusions are summarized in Section 7, with some thoughts given as to future work.

2. The CONDEL-2 Low-level Concurrent Machine Model - Operation and Characteristics

2.1. Hardware Description

The CONDEL-2 machine model was originally developed in [15]; other descriptions may be found in [17, 18, 19]. This machine is unusual in several respects, the most pervading being that it operates on the *static* instruction stream [12, 13, 22] of the input code.

In a static instruction stream representation the order of the code as it is examined for execution is that as it appears in memory, or in a program listing; this is in distinction to the classical order, the *dynamic* one, in which the order follows the value of the program counter. In static instruction stream machines a limited-size window on the code is kept from which

[2]The multi-CPU (i.e., multi- CONDEL-2) model also introduced in [19] is not considered herein; this is an area for further research.

instructions are issued for execution; the window holds a portion of the static instruction stream. In CONDEL-2, this window is called the *Instruction Queue (IQ)* [22]. See Figure 1. Since much of the instruction-issuing and execution portion of the machine is based on the Instruction Queue, its length n is a critical parameter.

All static instruction stream machines need some mechanism to keep track of the dynamic execution state of the instructions in the window. In CONDEL-2 this achieved by the *Advanced Execution (AE)* matrix [22], which keeps track of the state of execution of the instructions in the Instruction Queue. This matrix is a bit matrix[3] of dimension $n \times m$ in which the rows correspond to the instructions of the Instruction Queue, and the columns correspond to iterations or instances of the instructions; thus, when an instruction IQ_i is executed in iteration j, $AE_{i,j}$ is set to one.

In static instruction stream concurrent machines, it is necessary to know the ordering of the instructions, or the constraints on the ordering, namely the *dependencies* between the instructions. In CONDEL-2 the data and procedural (or branch) dependencies [18] are calculated at the same time the Instruction Queue is loaded, and are stored in $n \times n$ bit matrices; as an instruction is shifted into the Instruction Queue, the dependencies between this instruction and the prior $n-1$ instructions in the Queue are calculated and shifted diagonally into the dependency matrices. Note that these calculations, or their equivalent, are also determined by the compiler, as aids to optimizations. Although it would be possible to transmit this information to the machine, saving on the calculation hardware, the necessary instruction bandwidth would increase by about a factor of 5, which we prefer to avoid. The dependency information in CONDEL-2 is also used to update the *AE* matrix upon branch execution; thus the information serves many purposes and cannot easily be eliminated, without paying a significant cost or performance penalty. Therefore it is found to be beneficial to do the "same" thing twice, once in the software and once in the hardware. The overhead of doing this is not great.

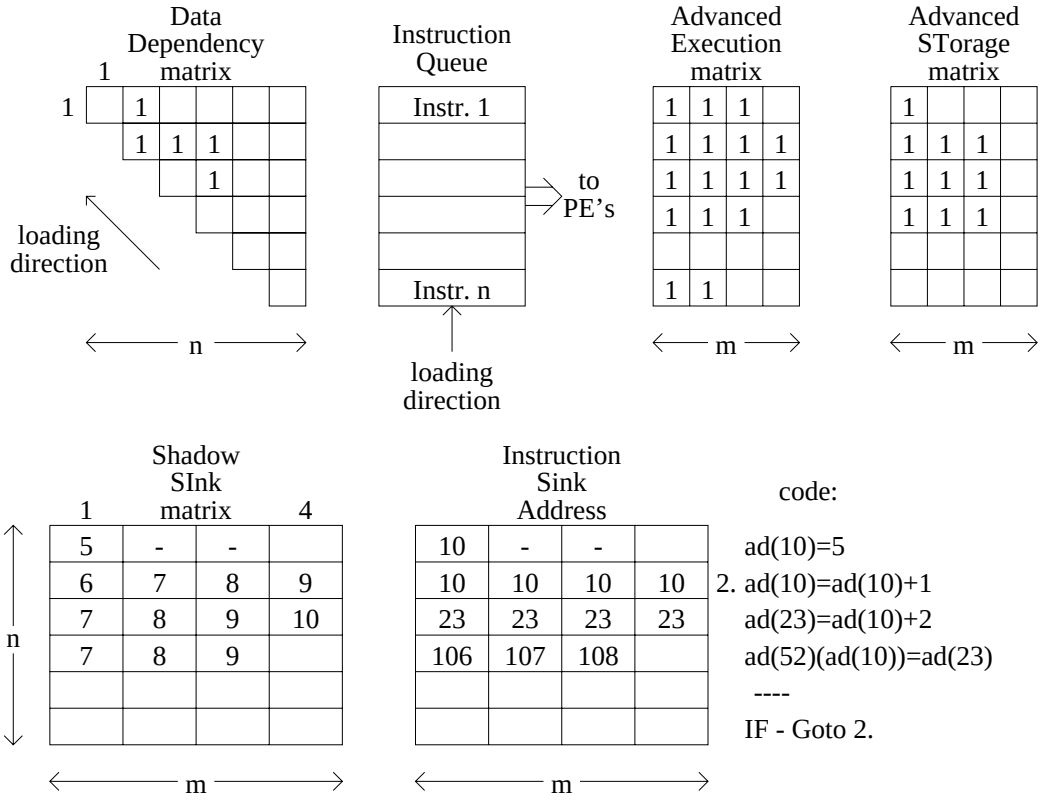
The following steps occur in the basic execution cycle of CONDEL-2:

1. if possible, load instructions into the Instruction Queue;
2. combine the dynamic execution state (Advanced Execution matrix contents) with the relatively static dependency state (e.g., Data Dependency matrix contents) to determine those instructions instances which can be executed concurrently in the current machine cycle;
3. issue these instructions for execution to either the branch execution unit or the processing elements;
4. simultaneously with step 3, update the dynamic execution state by setting the appropriate elements of the Advanced Execution matrix;
5. goto step 1.

There are three other concurrency structures of note in CONDEL-2; they have the same dimensions and correspondences as the Advanced Execution matrix. The *Shadow Sink* (*SSI*)

---

[3]This is a simplification of the actual system, in which *real* execution of an instruction instance is differentiated from its possible *virtual* execution [15, 16, 17, 18, 19]. For example, a virtual execution of an instance takes place when a forward branch is taken past the instruction.



Notes: "ad(A)" means the memory location at address "A".

Address 52 contains 100.

In the latest cycle, the second and third instructions have eagerly executed in iteration 4 (only 3 iterations have been enabled), the fourth instruction has executed in iteration 3, and the backward branch has executed in iteration 2. Also in this cycle, the second, third, and fourth instructions' sinks from iteration 3 have been written to memory. Since only 3 iterations have been enabled, no memory writes of sinks in iteration 4 take place, and the corresponding *AST* elements stay at 0.

Instruction 4 is an array write; see Section 2.2.

Figure 1. Basic CONDEL-2 concurrency structures, with example.

matrix holds the results or sink values of the corresponding instruction instances; the *Instruction Sink Address* (*ISA*) matrix holds the main memory addresses of the sinks held in the Shadow Sink matrix; both of these matrices are able to hold entire words per element. The *Advanced STorage (AST)* matrix is a bit matrix which indicates the main memory update status of the corresponding instruction instance; $AST_{i,j}=1$ indicates that the datum held in $SSI_{i,j}$ has been written into main memory location $ISA_{i,j}$, either really or virtually. This hardware is not essential for either static instruction stream or concurrent machines, but it is very beneficial.

These latter three structures allow the form of backward branch eager evaluation achieved in CONDEL-2 called *Super Advanced Execution*. An instruction may be executed ahead of time and its result and address stored in *SSI* and *ISA*, without modifying the machine state (*AST* is

held at 0, and main memory is not written). If it is determined that the instance is needed in the computation, then the writing of its sink to memory is allowed; otherwise, the result is simply ignored (not written to memory). This eager evaluation is effective only when an inner loop is completely held in the Instruction Queue, i.e., when *loop capture* has occurred.

The original version of the CONDEL-2 logic and structures had a hardware cost of $O(n^2m^2)$. A version of the machine in which only one instance per Instruction Queue row is allowed to execute in a cycle has cost $O(n^2m+nm^2)$; in practice, this is not a limiting constraint. The main point is that the hardware cost has a large dependence on n and m; since $m \leq n$, the cost can be approximated by $O(n^3)$. Thus, keeping the Instruction Queue short is a key concern in realizing the machine. From our prior work, a Queue length n of 32 seems to be a reasonable choice.

2.2. Constraints of the Hardware

Since the Instruction Queue length is limited, and since the best code execution (backward branch eager evaluation) occurs when loops[4] are captured, the loop size of input code should be kept less than $n = 32$.

For the following discussion, we note that the *domain* of a backward branch consists of the code between the branch and its target address, inclusive [15, 18]. Also, loops are formed by backward branches, possibly with other branches for exiting.

Given an unnested loop, code before and after the loop's backward branch's domain is executed only in one iteration. The code within the domain is executed (hopefully) in multiple iterations. In our machine model only one instance or iteration of an instruction is allowed to execute per cycle; and in *saturation* [15, 17, 18], a condition of peak performance, every instruction within a captured backward branch domain executes in one instance per cycle. Since this situation happens over many cycles (the number of iterations of the loop), over which the other instructions (before and after the loop) execute only once, it is desirable to maximize the size of the loop, such that it is less than or equal to n.

Another characteristic of the CONDEL-2 machine that constrains its operation is concerned with array and pointer accesses. In CONDEL-2, both types of accesses are of the form:

```
Array read:         Z = A(I)
Array write:        A(I) = Z

For pointers, array accesses are used with: I = 0
```

The effective address of the array element in question is determined by adding the contents of the location `I` to the contents of location `A`. Data dependencies between array accesses are only determined roughly, by comparing the "`A`" or Array Base address containers only; i.e., disambiguation is only performed at the array level. To guarantee the correct execution of code with such a system requires that all array writes to a particular array be performed before an array read to the same array is made; all array reads in CONDEL-2 are performed from main memory (scalar reads typically are made from the Shadow Sink matrix). This results in

[4] In particular, inner loops, but also outer loops.

anti-dependencies being enforced among array accesses to the same array. *Output* dependencies are not enforced, the composing writes being made independently to the Shadow Sink matrix.

There are two implications of the above, both resulting in overly restrictive dependencies. First, accesses to the same array but to different elements of the same array may unnecessarily be treated as data dependent accesses. Secondly, non-minimal (anti-) data dependencies will be enforced on all array accesses to the same array. There are partial solutions to these situations. If the compiler can determine that two accesses are to different elements of the same array, it can give the two accesses different Array Base address containers, i.e., different **A**'s, which hold the same Array Base address; therefore, CONDEL-2 will consider the two accesses to be independent, improving performance. For the second difficulty, the basic notion of providing multiple copies of a variable may be used [3] to eliminate the non-*flow* data dependencies. Since this is being applied to arrays, the extra storage required may be significant. These considerations are general and apply to any concurrent machine with a similar model of array accesses.

As this is a low-level concurrent machine, concurrency is exploited at a fine-grain level whenever the semantic dependencies allow. *Low-level spreading* [9], or *tree-height reduction* [8], consist of transforming an involved expression whose DAG forms an unbalanced tree to an algebraically equivalent expression whose DAG is more nearly a balanced tree; this reduces the height of the tree, hence reducing the number of time steps necessary to evaluate the tree concurrently. Low-level spreading has an effect on code execution in CONDEL-2, although not a great effect. If such a method is applied to the code within a loop, and the loop executes in saturation, then the only performance benefit will be a reduction in the computational latency (time to first result) and a slight reduction in the total execution time. A cost benefit is that the required value of *m* will be decreased, since fewer iterations are being (partially) executed at the same time. If the loop does not execute in saturation, but has a dependence cycle of length greater than one, then tree-height reduction may have a significant effect; if the reduction is applied to code in the critical path of the cycle, and said path length is reduced, then the computational bandwidth will be dramatically improved, since the execution time of such a loop is proportional to the dependence cycle length [19].

Another computation feature of interest is the way arithmetic reductions are and should be handled. A reduction is (e.g.) a summation of ν values, ν being the number of iterations of the loop. The summation typically has the form:

```
SUM = SUM + A(I).
```

If not paid attention to, such a form may conflict with other code transformations; we will return to this subject in the following section.

3. Desirable Code Transformations

In this section the desirable code transformations necessary to further enhance the low-level concurrent machine's performance are described [15]. In most cases, they are standard transformations or optimizations originally developed for other purposes [9]. In the examples, the code used has about a one-to-one correspondence with CONDEL-2 machine code.

3.1. Loop Unrolling

When an inner loop (backward branch domain) is less than half the length of the Instruction Queue, the loop may be unrolled one or more times to more fully utilize the CONDEL-2 processor resources. Unrolling can be a simple or complex procedure, depending on the type of loop involved.

If the number of iterations of a loop of size l is known at compile-time, then the body of the loop is copied $c=\lfloor(n-3)/(l-2)\rfloor-1$ times[5] and placed within one backward branch domain; the index incrementing instruction is also replicated c times, but is modified as shown in Figure 2. The "old loop index" instructions, `11.`, `21.` and `31`, generate individual indices for their respective replicated bodies. They all have the master loop index, `i`, as an input, and `i` is updated concurrently in `1.` This two-dimensional index updating, in space (different instructions), and time (different iterations), eliminates a potential dependency cycle which would occur if instructions `11.`, `21.` and `31.` were all of the form: `i = i + 1`. The temporaries do not need to be renamed, since CONDEL-2 eliminates shadow effects (antidependencies and output dependencies) occurring amongst scalars, i.e., multiple copies of `t1`, etc., exist.

If the number of iterations of a loop of size l is not known at compile-time, then the body of the loop is copied $c=\lfloor(n-1)/l\rfloor-1$ times as shown in Figure 3. There are now loop terminating branches for each loop body, at `17.`, `27.`, and `37.` Note that the loop bodies have been divided into roughly two parts: a part which can be eagerly evaluated (executed regardless of the state of some of the branches), e.g., `21.`, `22.`, `23.`, `24.`, and `26.`; and a part that cannot be moved since it potentially modifies important machine state (`D`), e.g., `24.'`, `25.`, and `27.` Instructions `14.`, `24.`, and `34.` are forward branch eagerly evaluated, since they have been moved out of the domain of the original forward branch. (The domain of a forward branch consists of the code from the branch to its target, exclusive.) With this splitting, some of the temporaries have to be renamed. The unrolled loop both executes in saturation, since there are no dependency cycles greater than 1; and executes with low computational latency, since in CONDEL-2 nested forward branches are independent [15, 18, 20], e.g., after `12.`, `16.`, `22.`, `26.`, `32.`, and `36.` execute in the second cycle, branches `14.'`, `17.`, `24.'`, `27.`, and `34.'` execute in the next cycle. These unrolling methods can be extended to general `WHILE` loops, although they may not always execute in saturation.

3.2. General Loop Fission

When a loop's length is greater than the size of the Instruction Queue, it is desirable to reduce the length of the loop so that Super Advanced Execution can take place, and the loop can be executed in saturation. The traditional loop fission-by-name [9] can be altered to a more general form as follows. If a loop is composed of independent iterations, then it can be fissioned into two or more loops with the use of temporary arrays; see Figure 4. The temporary array `Temp`

---

[5]The "2" and "3" represent the loop overheads of two instructions, one to test the ending condition, and one conditional backward branch to form the loop; and for "3", an additional instruction to update the master index (this could be eliminated, but is retained for clarity).

```
Original loop:
   0.         i = 0
   1. Loop:   i = i + 1              ; loop index
   2.         t1 = a(i)
   3.         t2 = b(i)
   4.         t3 = t1 + t2
   5.         d(i) = t3
   6.         c1 = (i < limit)       ; loop end test
   7.         if c1 goto Loop        ; backward branch


Loop unrolled twice, for three bodies; suitable for 18≤n≤23:
   11. Loop:  i1 = i + 1              ; old loop index
   12.        t1 = a(i1)
   13.        t2 = b(i1)
   14.        t3 = t1 + t2
   15.        d(i1) = t3

   21.        i2 = i + 2              ; old loop index
   22.        t1 = a(i2)
   23.        t2 = b(i2)
   24.        t3 = t1 + t2
   25.        d(i2) = t3

   31.        i3 = i + 3              ; old loop index
   32.        t1 = a(i3)
   33.        t2 = b(i3)
   34.        t3 = t1 + t2
   35.        d(i3) = t3

    1.        i = i + 3              ; new loop index

    6.        c1 = (i < limit)       ; loop end test
    7.        if c1 goto Loop        ; backward branch
```

Note: The loop has a fixed number of iterations, known at compile-time. This number of iterations (`limit`) is evenly divisible by $c+1=3$. Therefore only one exit conditional is needed (`7.`)

Figure 2. Easy loop unrolling.

holds all `limit` of the temporaries generated by loop 2a for use by loop 2b; therefore more data storage is necessary for the fissioned loops.

3.3. Array Aliasing

As described in the previous section, array aliasing may be performed on accesses which are determined not to be dependent on other accesses to the same array. See Figure 5 for an example; the time savings would be dramatically higher if the example code were within a loop, since in the original code case a dependency cycle of length 3 would be formed. Both simple and complex array subscript analysis, including the solving of diophantine equations, may be performed on array accesses to determine if array aliasing can be accomplished.

```
Original loop:
   0.        i = 0
   1. Loop:  i = i + 1              ; loop index
   2.        t1 = a(i)
   3.        t2 = b(i)
   4.'       if t1 goto Endif
   4.            t3 = t1 + t2
   5.            d(i) = t3
   6. Endif: c1 = (i < limit)       ; loop end test
   7.        if c1 goto Loop        ; backward branch


Loop unrolled twice, for three bodies; suitable for 25≤n≤33:
   0.        i = 0
   11. Loop:  i1 = i + 1              ; loop index
   12.        t11 = a(i1)
   13.        t2 = b(i1)
   14.            t31 = t11 + t2
   16.        c11 = (i1 < limit)      ; loop end test

   21.        i2 = i + 2              ; loop index
   22.        t12 = a(i2)
   23.        t2 = b(i2)
   24.            t32 = t12 + t2
   26.        c12 = (i2 < limit)      ; loop end test

   31.        i3 = i + 3              ; loop index
   32.        t13 = a(i3)
   33.        t2 = b(i3)
   34.            t33 = t13 + t2
   36.        c13 = (i3 < limit)      ; loop end test

    1.        i = i + 3               ; overall loop index

   14.'       if t11 goto Endif1
   15.            d(i1) = t31
   17. Endif1:   if ~c11 goto LoopExit  ; forward exit branch

   24.'       if t12 goto Endif2
   25.            d(i2) = t32
   27. Endif2:   if ~c12 goto LoopExit  ; forward exit branch

   34.'       if t13 goto Endif3
   35.            d(i3) = t33
   37. Endif3:   if c13 goto Loop       ; backward branch

    9. LoopExit:  -----------
```

Note: The number of iterations is not known at compile time.

Figure 3. Hard loop unrolling, with partial forward branch eager evaluation.

```
Original loop, suitable for n≥13:
   0.         i = 0
   1. Loop:   i = i + 1              ; loop index
   2.         t1 = a(i)
   3.         t2 = b(i)
   4.         t3 = t1 + t2
   5.         t4 = t1 * t2
   6.         t5 = t3 / t4
   7.         t6 = t5 + 4
   8.         t7 = t6 * 10
   9.         t8 = d(i)
   10.        t9 = t7 + t8
   11.        d(i) = t9
   12.        c1 = (i < limit)       ; loop end test
   13.        if c1 goto Loop        ; backward branch


Fissioned loops, suitable for n≥9:
   0.         i = 0
   1. Loop2a: i = i + 1              ; loop index
   2.         t1 = a(i)
   3.         t2 = b(i)
   4.         t3 = t1 + t2
   5.         t4 = t1 * t2
   6.         t5 = t3 / t4
   6.'        Temp(i) = t5           ; save intermediate result
   12.        c1 = (i < limit)       ; loop end test
   13.        if c1 goto Loop2a      ; backward branch

   0.         i = 0
   1. Loop2b: i = i + 1              ; loop index
   6.         t5 = Temp(i)           ; get intermediate result
   7.         t6 = t5 + 4
   8.         t7 = t6 * 10
   9.         t8 = d(i)
   10.        t9 = t7 + t8
   11.        d(i) = t9
   12.        c1 = (i < limit)       ; loop end test
   13.        if c1 goto Loop2b      ; backward branch
```

Figure 4. Example of loop fission.

3.4. Array Shadow Effects Elimination

This is also known [9] as "array renaming". The aim is to eliminate anti- and output dependencies amongst array accesses. In CONDEL-2, it is only necessary to eliminate the anti-dependencies with the compiler, as output dependencies are not enforced by the hardware. This also requires subscript analysis, possibly of a complex nature. See Figure 6 for an example.

```
Original code:
   1.        t1 = a(i)
   2.        i1 = i + 1
   3.        a(i1) = k
   4.        i2 = i + 2
   5.        t2 = a(i2)

Modified code, with array write aliased:
   0.        aa = a
   1.        t1 = a(i)
   2.        i1 = i + 1
   3.        aa(i1) = k
   4.        i2 = i + 2
   5.        t2 = a(i2)
```

Notes: In the original code, **3.** is dependent on **1.**, and **5.** is dependent on **3.**; therefore, the code as written takes three cycles to execute.

In the modified code, the array write is made to the aliased version of the same array **a**; instruction **0.** gives **aa** the same array base address as **a**. Now there are no array dependencies, and the code executes in two cycles.

Figure 5. Example of array aliasing.

```
Original code:
   1.        i1 = i + 1
   2.        t1 = a(i1)
   3.        a(i1) = k
   4.        i2 = i + 1
   5.        t2 = a(i2)

Modified code, with anti-dependency removed via renaming:
   1.        i1 = i + 1
   2.        t1 = a(i1)
   3.        a2(i1) = k
   4.        i2 = i + 1
   5.        t2 = a2(i2)
```

Notes: The orginal code takes 4 cycles to execute.

In the modified code, **a** and **a2** are different arrays. The modified code takes 3 cycles to execute.

Figure 6. Example of array shadow effects elimination.

3.5. Tree-Height Reduction

Tree-height reduction [8] is well known; it is also known as "low-level spreading" [9]. It can reduce the computational latency of code executed on CONDEL-2. It is particularly useful in CONDEL-2 when applied to the members of a dependence cycle, as it can result in the shrinking of the cycle length. Being so common, this transformation was applied to the control versions of

the loops as well as the optimized loops.

3.6. Dependence Cycle Reduction

This can be be accomplished by tree-height reduction as mentioned above, but it can also be accomplished, and perhaps in a better fashion, by paying attention to the critical path of the cycle; see Figure 7.

```
Original loop, cycle length is 3:
   0.        i = 0
   1. Loop:        t0 = a(i)
   2.              t1 = t0 + z
   3.              t2 = c + d
   4.              t3 = t1 + t2
   5.              t4 = e + 5
   6.              z = t3 + t4
   7.              i = i + 1
   8. Endloop: -----------

Modified loop, associativity of "+" used, cycle length is 1:
   0.        i = 0
   1. Loop:        t0 = a(i)
   3.              t2 = c + d
   4.              t3 = t0 + t2
   5.              t4 = e + 5
   6.              t5 = t3 + t4
   6.'             z = z + t5
   7.              i = i + 1
   8. Endloop: -----------
```

Note: In both loops the critical dependence cycle path is through the variable **z**.

Figure 7. Dependence cycle length reduction example.

3.7. General Arithmetic Reduction Considerations

Computing the sum or product of a series of numbers, possibly elements of an array, is a common loop calculation. If realized in unrolled code in an unthinking manner, dependence cycles greater than length 1 may result, causing less than peak performance to be exhibited by the code. The situation is analogous to the maintenance of the overall loop index in the unrolled code at the beginning of this section; the solution is also similar, see Figure 8. In the example, summation is performed concurrently along both the space and time dimensions. Tree-height reduction is applied to the partial sums within an iteration.

3.8. Forwarding

In this transformation duplicate array reads are eliminated, forwarding the result of the first read to subsequent reads, thereby eliminating some array accesses and their accompanying dependencies.

```
Original, rolled loop:
   0.      i = 0
   0.5     sum = 0
   1. Loop:  i = i + 1
   2.        t = a(i)
   3.        sum = sum + t
   4.        if "not done" goto Loop


Unrolled loop, sum calculation forms cycle of length 3:
   0.      i = 0
   0.5     sum = 0

   11. Loop: i1 = i + 1
   12.       t = a(i1)
   13.       sum = sum + t

   21.       i2 = i + 2
   22.       t = a(i2)
   23.       sum = sum + t

   31.       i = i + 3
   32.       t = a(i)
   33.       sum = sum + t

   4.        if "not done" goto Loop


Unrolled loop, proper sum calculation made, cycle is of length 1:
   0.      i = 0
   0.5     sum = 0

   11. Loop: i1 = i + 1
   12.       t1 = a(i1)

   21.       i2 = i + 2
   22.       t = a(i2)
   23.       ts = t1 + t

   31.       i = i + 3
   32.       t = a(i)
   33.       sumiteration = ts + t      ; compute partial sum of iteration

   43.       sum = sum + sumiteration   ; compute overall sum

   4.        if "not done" goto Loop
```

Figure 8. Examples of dependence cycle lessening in arithmetic reductions.

3.9. Summary and Comments

The desirable code transformations for the static instruction stream concurrent machine have been presented and discussed. The transformations of Loop Unrolling and General Loop Fission are applicable to static instruction stream machines in general, not just CONDEL-2. The remaining transformations are applicable to concurrent machines, also in general.

4. Experiment Description

Two sets of benchmark programs were used in the experiments. The Scientific Set consists of the 14 original Lawrence Livermore Loops [10] with the number of iterations reduced by about a factor of 10, to reduce simulation time; this makes our results more conservative, if anything, since executing more iterations would reduce the effect of startup transients. The General Purpose Set consists of ten benchmarks chosen for their variety of computation; descriptions may be found in [15, 18].

As a control, the benchmarks were left as they were after originally being encoded in CONDEL Assembly language for simulation; the results of CONDEL-2 with the unoptimized benchmarks were obtained from prior work [15].

The code transformations described in the last section were applied by hand to all of the benchmarks, generating code optimized for CONDEL-2. The specific optimizations applied to each benchmark are tabulated in Section 5. Their execution was then simulated on a CONDEL-2 functional simulator [15]. The results of the simulations, and comparisons to the unoptimized execution times, are given in Section 6.

5. Lexical Effects of Code Transformations

The effects on loop size and number of the code transformations are shown in Table 1. Also shown in the table are indications of which transformations were applied to which benchmarks. For the Livermore Loops, all of the loops except for loops 5 and 6 were able to be transformed in some fashion. For the General Purpose benchmarks only about half were able to be transformed with some hope of performance improvement. The ACM410, ACM428 and Getblk benchmarks consist of unstructured code, which proved resistant to this researcher's hand optimizations; an automated method might do better. The Dhrystone benchmark was left as it was since its loops only execute once or twice; it is questionable as to how realistic this is.

6. Performance Effects of Code Transformations

In this section the performance effects of the code transformations are given and analyzed.

For each benchmark, the execution time[6] was obtained for both the untransformed code and the code optimized via the transformations given in Section 5, for four values of the Advanced Execution matrix width, *m*. *m* roughly corresponds to the maximum number of iterations able to be active at any time during code execution. The results are shown in Figures 9 and 10.

---

[6]The execution times were measured assuming a 0 cost for loading the Instruction Queue. Taking this into account could potentially alter the results; see [19] for an analysis. Note that in this work, the code is tailored to the machine, i.e., the loops are not larger than the Instruction Queue, thereby reducing the number of load cycles required, in the case of a loop upon which loop fission has been performed. The effects of unrolling loops are to increase the Queue loading overhead, but it should be relatively slight, as pipelining can be used to ameliorate the effects, which are simple due to the lack of adverse control flow in such situations.

Benchmark	Loop size(s) (before/after transform.) ***	Loop Unrolling ***,*4	General Loop Fission	Array Aliasing	Array Shadow Effects Elim.	Depend. Cycle Reduc.	Arith. Reduc.	Forward.
LLL 1	14/26	X - 2						
LLL 2	31					X		
LLL 3	6/32	X - 6				X	X	
LLL 4**	12,23/14,27	X - 2,1						
LLL 5*								
LLL 6*								
LLL 7	35/20;29	X - 2;1	X					
LLL 8	158/31;32;25; 26;25;26;25;26	X - 1;2;1; 1;1;1;1;1	X	X				
LLL 9	53/28;32		X	X				
LLL 10	72/29;26;28		X		X			
LLL 11	8/31	X - 6				X	X	
LLL 12	8/32	X - 5						
LLL 13	86/29;27;29		X	X				X
LLL 14	35/18;18		X	X				X
ACM410*								
ACM428*								
BCDBin	9,14/29*5	X - 4,1				X	X	
Dhrystone*								
Getblk*								
Hardshuffle	8;7,17;7/ 29;16,28;28	X - 5;2,1;5		X*7				
MCF4	47/29;28		X	X				
MCF7*6	7;4/30;29	X - 5;7						
Puzzle	10;9;7;9;27/ 31;28;29;28;27	X*8 - 3;3;4;3;1						
Shellsort	13/27	X*8 - 2						

Notes: An "X" in a column indicates that the corresponding transformation was applied to the benchmark.

* These benchmarks did not undergo any transformations, as they were either not applicable or they would not have helped.

** Other optimizations applied to this benchmark were the substitution of scalar references for array references, and the moving of loop invariant code out of the loop.

*** Commas separate nested loops, semicolons separate disjoint loops.

*4 The numbers after the "X"'s indicate the number of loop bodies in the unrolled code.

*5 The inner loop was eliminated by the unrolling.

*6 Also, loop invariant code was moved out of the first loop.

*7 Array accesses were synchronized by their index references.

*8 The forward branch eager evaluation optimization was performed on these benchmarks.

Table 1. Transformation data for the benchmarks.

The three plots per graph shown in the figures are as follows:

- Plot 1 (solid line): These speedups correspond to an unoptimized benchmark executed on the CONDEL-2[7] machine, as compared to the standard time to execute the benchmark on a strictly sequential version of CONDEL-2, i.e., a sequential machine with the CONDEL-2 instruction set.
- Plot 2 (dotted line): These values of *S* are obtained by dividing the *best* sequential time of the benchmark by the time to execute the optimized code concurrently on CONDEL-2. The sequential time used was for either the optimized or unoptimized benchmark, whichever was faster.
- Plot 3 (dashed line): These speedups are computed from the *standard* sequential execution time divided by the time to execute the optimized version of the benchmark. This plot demonstrates the total effect of the optimizations. This is reasonable, as one might not use the optimized version of the code on a sequential machine since it may use significantly more storage.

All 12 of the optimized Livermore Loops achieved significant performance improvements with the code transformations. In 10 of these, loops 1, 2, 3, 4, 7, 8, 9, 10, 11, and 12, the code was executed in saturation. The overall effect of the code transformations was to speedup execution[8] by a factor of 3. Most of the performance gains were obtained with $m=8$, which should be realizable.

The results were less gratifying for the General Purpose benchmark set, but were still quite significant. Most of the optimized codes exhibited large gains with the code transformations; however, one benchmark, Shellsort, was optimized, but did not demonstrate a significant gain. The overall effect of the optimizations was to speedup the benchmarks' execution by a factor of about 2. As with the Scientific Set, with $m=8$ most of the gains were obtained.

7. Conclusions

In this paper we have described several code transformations which are relatively straightforward for a compiler to perform, many of which are well known, and all of which were applied to a somewhat different environment, namely a concurrent static instruction stream machine. It was found that many of the existing code transformations also work for static instruction stream concurrent machines, but in some cases, e.g., loop fission, for different reasons. Although applied to a specific machine, the transformations are applicable to both static instructions stream and concurrent machines in general. The lexical and performance effects of the transformations were given, demonstrating gains due to the transformations of a factor of 2 or 3, leading to overall speedups of about 15 for a Scientific Set of benchmarks, and about 6 for a General Purpose Set. Software enhancements to input code can therefore significantly improve overall concurrent system performance.

In the future, we desire to construct a compiler embodying these and possibly other code

---

[7]For readers of some of our prior work, this version of the machine corresponds to: *dct*=3, *pct*=C.

[8]We use the arithmetic mean here since we believe it gives a user a better idea of how other code will execute, given that there is no implied weighting of the benchmarks.

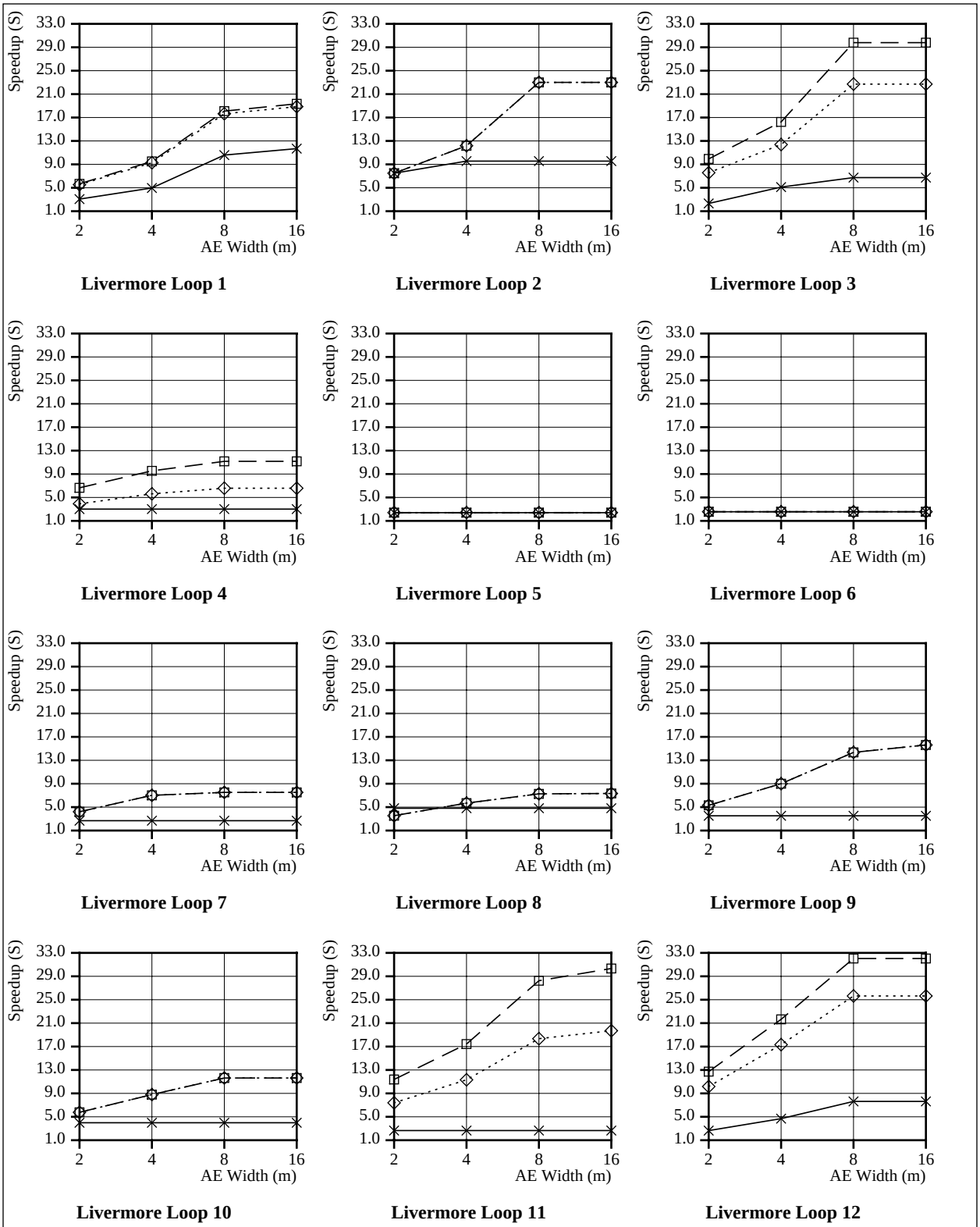


Figure 9. Scientific Set performance results; *n*=32. (cont. on next page)

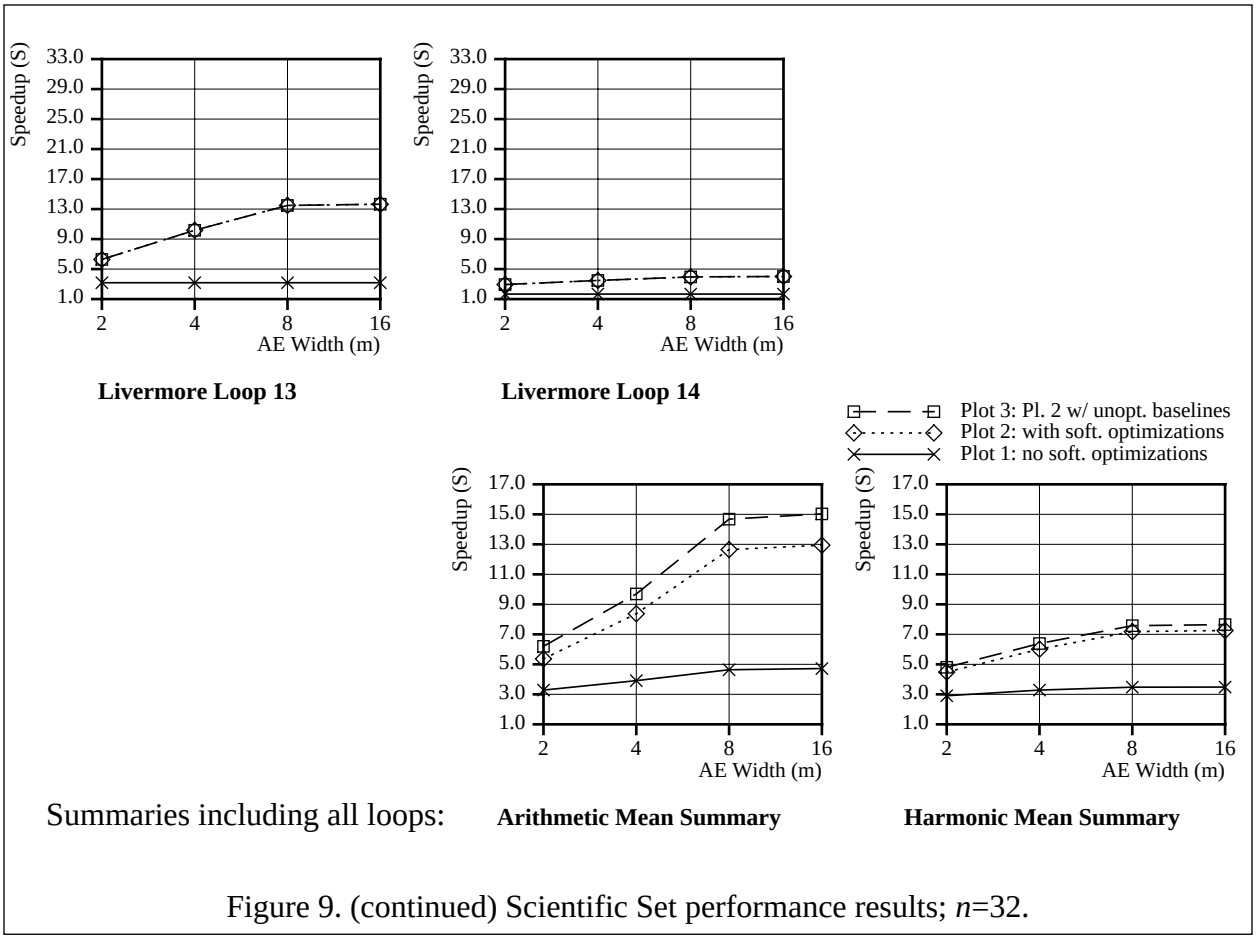


Figure 9. (continued) Scientific Set performance results; *n*=32.

optimizations, as well as to construct an improved CONDEL-2 processor.

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* (September 1986), 815-828.

2. Aiken, A. and Nicolau, A. Perfect Pipelining: A New Loop Parallelization Technique. Proceedings of the 1988 European Symposium on Programming, , 1988. Also available as Dept. of Computer Science Technical Report Number 87-873, Cornell University, Ithaca, N.Y. 14853.

3. Chamberlin, D. D. The Single-Assignment Approach to Parallel Processing. Fall Joint Computer Conference, AFIPS, 1971, pp. 263-269.

4. Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B. and Rodman, P. K. "A VLIW Architecture for a Trace Scheduling Compiler". *IEEE Transactions on Computers C-37*, 8 (August 1988), 967-979.

5. Ebcioglu, K. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. Proceedings of the Twentieth Annual Workshop on Microprogramming (MICRO-20), Association of Computing Machinery, December, 1987, pp. 69-79.

6. Ebcioglu, K. Personal communication. .

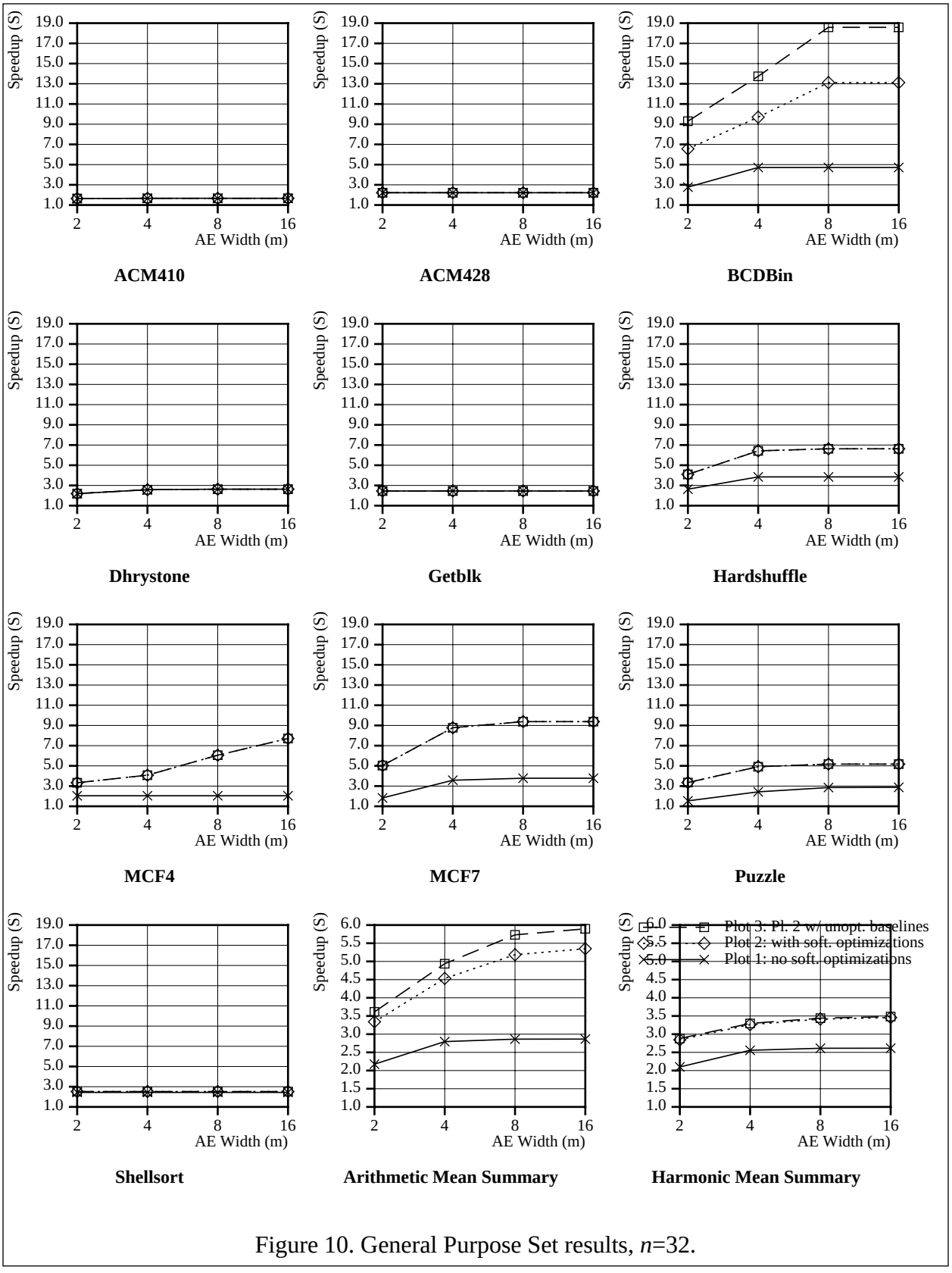


Figure 10. General Purpose Set results, *n*=32.

7. Fisher, J. A. Very Long Instruction Word Architectures and the ELI-512. Proceedings of the 10th Annual International Symposium on Computer Architecture, ACM-SIGARCH and the IEEE Computer Society, June, 1983, pp. 140-150.
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 (December 1972), 1293-1310.
9. Padua, D. A. and Wolfe, M. J. "Advanced Compiler Optimizations for Supercomputers". *Communications of the ACM 29*, 12 (December 1986), 1184-1201.
10. Riganati, J. P. and Schneck, P. B. "Supercomputing". *COMPUTER, IEEE Computer Society 17*, 10 (October 1984), 97-113.
11. Thorton, J. E. Parallel Operation in the Control Data 6600. Proceedings of the Fall Joint Computer Conference, AFIPS, 1964, pp. 33-40.
12. Tjaden, G. S. *Representation and Detection of Concurrency Using Ordering Matrices*. Ph.D. Th., The Johns Hopkins University, 1972.
13. Tjaden, G. S. and Flynn, M. J. "Representation of Concurrency with Ordering Matrices". *IEEE Transactions on Computers C-22*, 8 (August 1973), 752-761.
14. Tomasulo, R. M. "An Efficient Algorithm for Expoiting Multiple Arithmetic Units". *IBM Journal* (January 1967), 25-33.
15. Uht, A. K. *Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams*. Ph.D. Th., Carnegie-Mellon University, Pittsburgh, PA, December 1985. Available from University Microfilms International, Ann Arbor, Michigan, U.S.A..
16. Uht, A. K. An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code. 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.
17. Uht, A. K. and Wedig, R. G. Hardware Extraction of Low-level Concurrency from Serial Instruction Streams. Proceedings of the International Conference on Parallel Processing, IEEE Computer Society and the Association for Computing Machinery, August, 1986, pp. 729-736.
18. Uht, A. K. Incremental Performance Contributions of Hardware Concurrency Extraction Techniques. 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..
19. Uht, A. K., Polychronopoulos, C. D., and Kolen, J. F. On the Combination of Hardware and Software Concurrency Extraction Methods. Proceedings of the Twentieth Annual Workshop on Microprogramming (MICRO-20), Association of Computing Machinery, December, 1987, pp. 133-141.
20. Uht, A.K. "A Theory of Reduced and Minimal Procedural Dependencies". *IEEE Transactions on Computers* (Submitted: December 1988). Under review..
21. Uht, A. K. "Concurrency Extraction via a Hardware Method Executing the Static Instruction Stream". *IEEE Transactions on Computers* (Submitted: March 1989). Under review..
22. Wedig, R. G. *Detection of Concurrency in Directly Executed Language Instruction Streams*. Ph.D. Th., Stanford University, June 1982.

Author's Address

Dr. Augustus K. Uht; Department of Computer Science and Engineering, C-014; University of California, San Diego; La Jolla, CA 92093.

Table of Contents

List of Figures

List of Tables