Parallel Computing, Vol. 20, No. 5, May 1994, pp. 729-751

# A Multiprocessor Architecture Combining Fine-Grained and Coarse-Grained Parallelism Strategies

David J. Lilja
Department of Electrical Engineering
University of Minnesota
200 Union Street S.E.
Minneapolis, MN 55455

Phone: (612) 625-5007
FAX: (612) 625-4583
E-mail: lilja@ee.umn.edu

November 10, 1992
Revised: May 26, 1993

(to appear in)
*Parallel Computing*

**Abstract**

A wide variety of computer architectures have been proposed that attempt to exploit parallelism at different granularities. For example, pipelined processors and multiple instruction issue processors exploit the fine-grained parallelism available at the machine instruction level, while shared memory multiprocessors exploit the coarse-grained parallelism available at the loop level. Using a register-transfer level simulation methodology, this paper examines the performance of a multiprocessor architecture that combines both coarse-grained and fine-grained parallelism strategies to minimize the execution time of a single application program. These simulations indicate that the best system performance is obtained by using a mix of fine-grained and coarse-grained parallelism in which any number of processors can be used, but each processor should be pipelined to a degree of 2 to 4, or each should be capable of issuing from 2 to 4 instructions per cycle. These results suggest that current high-performance microprocessors, which typically can have 2 to 4 instructions simultaneously executing, may provide excellent components with which to construct a multiprocessor system.

**Keywords:** coarse-grained; fine-grained; instruction-level parallelism; loop-level parallelism; performance comparisons; pipelining; multiprocessor; superscalar.

## 1. Introduction

There are two basic techniques that have been used to increase computer performance — 1) implement the machines in faster technology, or 2) perform more operations in parallel. These techniques are not mutually exclusive, but as semiconductor technology has matured it has become apparent that more parallelism must be exploited in order to keep increasing system performance. A wide variety of architectures have been proposed that attempt to exploit the parallelism available in application programs at different granularities [20]. For example, pipelined processors [1, 11, 26] and multiple instruction issuing processors, such as the superscalar [11, 12, 32, 33] and VLIW [5, 8, 18] machines, exploit the fine-grained parallelism available at the machine instruction level. In contrast, shared memory multiprocessors [9, 14, 24] typically exploit coarse-grained parallelism by distributing entire loop iterations to different processors.

Each of these parallel architectures have significant differences in synchronization overhead, instruction scheduling constraints, memory latencies, and implementation details, making it difficult to determine which architecture is best able to exploit the parallelism available in an application program. In addition, several high-performance microprocessors have recently been announced that are capable of simultaneously executing two to four independent operations. These microprocessors may provide excellent building blocks for constructing a large-scale multiprocessor that can exploit parallelism at several different granularities simultaneously. To maximize the performance of this type of ‘‘multigrained’’ architecture, however, the best mix of fine-grained and coarse-grained parallelism must determined. That is, the degree of fine-grained parallelism needed within each of the individual processors must be determined in conjunction with the total number of processors to be used in the system.

This paper uses a register transfer level simulation methodology to examine the performance of a pipelined multiprocessor architecture and a superscalar multiprocessor architecture that combine capabilities for exploiting both fine-grained and coarse-grained parallelism from a single application program. Section 2 reviews some relevant background material and surveys previous work comparing pipelined and superscalar processors. The machine models and simulation methodology used in this study are described in Section 3. Section 4 presents measurements of the maximum inherent parallelism of the test programs. This section also evaluates the performance impact of several important factors that affect the architecture of the fine-grained and coarse-grained processors. Section 5 then presents simulation results to determine the degree of fine-grained parallelism that should be used in each of the individual processing elements in a coarse-grained multiprocessor system to achieve the best performance when executing the test programs used in this study. The results and conclusions are summarized in the last section.

## 2. Background and Related Work

This section briefly discusses the inherent limitations to exploiting the parallelism available in application programs. It also reviews previous studies measuring how much parallelism actually is available, and studies comparing the performance of multiple instruction issue processors and pipelined processors.

### 2.1. Limitations to Parallelism

The parallelism available in an application program is limited by its dependences. A dependence between two operations is a conflict that prevents the operations from executing concurrently. Dependences can be categorized into three types: resource, control, and data. A

*resource dependence* exists between two operations when they both need to use the same physical resource at the same time. These dependences are a physical limitation of the actual machine on which a program is to be run. A *control dependence* occurs when an operation should be executed only if a previous operation produces a certain value. For instance, a conditional operation produces a control dependence to a different operation if the second operation is to be executed only if the condition evaluates to a specified value.

*Data dependences,* also known as *hazards,* are read-write conflicts between two operations in which they both access the same storage location, such as a register or a location in memory. In a *flow dependence* (read-after-write hazard), one operation needs a value generated by a previous operation before the latter operation can begin executing. An *output* dependence (write-after-write hazard) occurs between two operations when they both write to the same storage location. The correct ordering of these operations is required to ensure that any intervening operations that read this location obtain the correct value before it is overwritten by the second operation. An *antidependence* (write-after-read hazard) exists when a later operation may overwrite a value still waiting to be read by an earlier operation. Both antidependences and output dependences occur when variable names and registers are reused by the programmer or by the compiler to reduce the number of unique memory locations referenced by a program. They can be eliminated by renaming variables so that a unique *value* of a variable has a unique *name.* The cost of this renaming is a potentially large increase in the memory requirements of the program [15].

### 2.2. Available Parallelism

Several studies have attempted to determine how much parallelism is actually available in application programs [2, 3, 12, 16, 19, 22, 25, 28, 30, 33, 35, 36]. These studies have examined a wide variety of numeric and non-numeric application programs and have measured speedups ranging from slightly more than one to as much as several thousand when ignoring all resource dependences [20]. In all cases, these studies indicate that maximum speedups of only two to four are possible when limiting parallelism extraction to a single basic block. When basic block boundaries are ignored so that the entire program is available for extracting parallelism, however, numeric engineering and scientific programs typically have a high level of inherent parallelism. Computation which is less structured than these numeric applications has relatively little parallelism, even with infinite resources available.

In addition to measuring the maximum amount of parallelism available in application programs, several studies have investigated the interaction of multiple instruction issuing with pipelining [10, 12, 31]. This work has shown that at the basic block level, pipelining and multiple instruction issuing are essentially equivalent in their abilities to exploit fine-grained parallelism. Indeed, the Astronautics ZS-1 [29] and the SIMP (Single Instruction stream/Multiple instruction Pipelining) processor [21] both implement multiple independent execution pipelines to exploit the advantages of both types of architectures. This previous work has not studied the interaction of fine-grained parallelism strategies with coarse-grained strategies, however. Consequently, the experiments presented in this paper extend this previous work by examining a coarse-grained shared memory multiprocessor in which each individual processor can exploit fine-grained parallelism through pipelining or multiple instruction issuing. The primary goal of these experiments is to determine the mix of granularities that will produce the highest performance.

## 3. Simulation Methodology

Register-transfer level simulations of a pipelined processor, a superscalar processor, and a shared memory multiprocessor are used to examine the capabilities of each individual architecture to exploit the parallelism available in several computation-intensive scientific and engineering application programs. These individual models then are combined into a single model of a pipelined superscalar multiprocessor. This section describes these machine models and the simulation parameters used in this study.

### 3.1. Machine Models

To provide a common point of comparison for the different parallel architectures, a *basis processor* is defined that is capable of issuing one instruction per cycle to one of four different functional units: a floating point unit, an integer unit, a miscellaneous unit, and a memory controller. The memory load instruction is nonblocking so that there may be multiple outstanding memory read requests, and write operations are buffered to prevent them from stalling the processor. The instruction fetch and decode stages of the basis machine are assumed to be pipelined and to completely overlap with other instruction execution. As a result, the latencies for only the execution stage of the pipeline, shown in Table 1, are simulated.

The multiprocessor consists of $p$ copies of the basis machine connected to a shared memory via a multistage interconnection network. Memory delays for references that are not satisfied by the data cache are modeled using the equation $T_{mem} = a_0+a_1\log_2 p+f(\log_2 p, util)$. The constant term, $a_0$, is the interface delay between the processor, the network, and the global memory, including the access delay in the memory module itself. The $a_1\log_2 p$ term represents the primary delay within the network due to the switching elements. Contention delays, modeled by the last term, are a function of the number of stages in the network ($\log_2 p$), and the utilization of the network. It is assumed that these delays are about 50 percent of the network delay [13, 23]; i.e., the last term is $0.5a_1\log_2 p$. Coefficient values are loosely based on the Cedar system [14] with $a_0$=17 and $a_1$=1, giving $T_{read} = 17+\lceil 1.5\log_2 p \rceil$ cycles.

Parallel loop iterations in the multiprocessor are statically scheduled across the $p$ processors using a Doacross model [6]. Figure 1(a) shows a parallel loop that has a lexically backward dependence, through variable $B$, in which statement $S_1$ in iteration $i$ must execute after statement $S_2$ in the previous iteration, $i-1$. As shown in Figure 1(b), the start of each iteration in processor 0 is

Table 1: Basis machine instruction latencies.
($l_i$, in cycles)

| Operation | Functional Unit | | | |
|---|---|---|---|---|
| | Integer | Float | Misc | Memory |
| add/subtract | 1 | 1 | | |
| multiply | 1 | 2 | | |
| divide | 2 | 6 | | |
| transcendental | | 6 | | |
| branch | | | 1 | |
| misc | 1 | 2 | 1 | |
| load/store | | | | (see text) |

```
DO i=1,N
  S1:  A(i) = B(i-1) + X
  S2:  B(i) = A(i) + Z(i-1)
  S3:  C(i) = A(i) + Y
  S4:  D(i) = A(i) * B(i)
ENDDO
```

(a) Loop with inter-iteration dependences.

| | $P_0$ | $P_1$ | $P_2$ |
|---|---|---|---|
| $T_1$ | $S_1(1)$ | | |
| | $S_2(1)$ | | |
| $T_2$ | $S_3(1)$ | $S_1(2)$ | |
| | $S_4(1)$ | $S_2(2)$ | |
| $T_3$ | | $S_3(2)$ | $S_1(3)$ |
| | | $S_4(2)$ | $S_2(3)$ |
| $T_4$ | $S_1(4)$ | | $S_3(3)$ |
| | $S_2(4)$ | | $S_4(3)$ |
| | $S_3(4)$ | | |
| | $S_4(4)$ | | |

(b) Synchronization delays.

Figure 1: Example Doacross loop.

delayed by $T_{delay}=T_4-T_3$ cycles due to the synchronization with the other processors. The total synchronization time for each iteration is $(p-1)T_c$, where $T_c=T_2-T_1$. Since the processor can potentially overlap some of its execution with the synchronization operations, it is actually delayed by $T_{delay} = \max\left[0, (p-1)T_c - T_{overlap}\right]$ cycles during each iteration, where $T_{overlap}=T_3-T_2$. The *max* operation is required for the case when $T_{overlap}$ is actually longer than the synchronization delay. The time $T_{delay}$ is added to the total simulation time for each iteration executed by processor 0. Since processor 0 also executes all of the serial code between parallel loops, its final execution time is the total time required by the multiprocessor to execute the application program.

In the model of the superscalar processor, a single instruction issuing unit examines the dependences between the instructions available in the instruction prefetch buffer and the instructions currently executing. Each cycle, the first group of $j$ instructions that have no dependences among themselves, nor with the currently executing instructions, are issued to the $j$ available functional units. If fewer than $j$ instructions are available to be issued, *no-ops* are issued to the idle functional units. The instruction latencies for each type of instruction are the same as those in the basis machine, as shown in Table 1.

In the pipelined processor, one instruction is issued per cycle, but the cycle time of the $m$-stage pipelined machine is reduced by a factor of $m$ compared to the basis machine. This reduction in cycle time allows instructions to issue up to $m$ times faster than in the basis machine. If there is a dependence between the next instruction waiting to be issued and currently executing instructions,

*no-ops* are issued until the instruction causing the dependence completes its execution. The number of cycles required to produce a result (i.e. the latencies in Table 1) are multiplied by $m$ for the pipelined processor so that each instruction still takes the same amount of absolute time to complete. In the following simulations, the degree of pipelining is varied from 1 to 32. Due to propagation delays and latch overhead, however, there is a limit to how finely a pipeline can be divided. In fact, these implementation constraints limit actual pipelines to roughly six stages for an optimal degree of pipelining [7, 17]. These implementation effects are not taken into account in this study so that the simulations can focus on differences in architectural parallelism. These results then must be considered in light of what can actually be implemented.

Because there is only a single program counter in both the superscalar and the pipelined processors, there are no synchronization delays in either of these processors. Since all $j$ of the instructions issued in any cycle in the superscalar processor could be memory read operations, it is assumed that its delay when referencing noncached data is similar to the delay in the multiprocessor, giving $T_{read}=17+\lceil 1.5\log_2 j \rceil$ cycles. It is assumed that the memory in the pipelined processor can be pipelined to the same degree as the processor giving its memory latency as $T_{read}=17m$ cycles when referencing noncached data. With this memory delay model, the latency to read memory in the pipelined processor is constant relative to the basis machine, but it can accept requests up to $m$ times faster. It is assumed that the latency of the processor registers and the cache system improves directly with increases in both $j$ and $m$. For instance, the number of ports in the register file in the superscalar processor increases with $j$. Also, the time required to access a register in the pipelined processor reduces in proportion to $m$ so that all register accesses require only a single cycle as the degree of pipelining increases. These register access assumptions are probably reasonable for moderate values of $j$ and $m$, say less than around six, but improvements in the register access time cannot continue indefinitely with large values of $j$ and $m$. These simulations thus provide an optimistic bound on the performance of the fine-grained processors.

A shared memory multiprocessor consisting of $p$ copies of a pipelined superscalar processor also can be simulated. The memory delay for accessing noncached data in this type of system is $T_{read}=m[17+\lceil 1.5\log_2 pj \rceil]$ cycles. With this model, the memory delay increases logarithmically with the product of the number of processors and the degree of superscalar parallelism within each processor. As the degree of pipelining within each processor increases, the memory latency stays constant relative to the basis machine, but the system's ability to service memory requests increases by a factor of $m$. As in the pipelined processor, the clock cycle of the processors in this combined system is reduced by a factor of $m$ while the execution time latencies are again multiplied by $m$. This simulation model thereby combines the parameters of all three architectures.

### 3.2. Simulation Technique

Starting with Fortran source code, a parallelizing compiler is used to generate the corresponding parallel assembly code. This assembly code is converted into equivalent assembly code for the basis machine which then is executed by a register transfer level processor emulator to produce a complete instruction trace for one processor of a $p$ processor system. The trace is generated for the processor that executes all of the serial code between parallel loops in addition to its share of the parallel loop iterations. A timing simulator analyzes the dependences in the instruction trace and determines the execution time for the given values of $p$, $j$, and $m$ using the instruction latencies and the memory and synchronization delays described in the previous section. All instructions are executed out of an instruction cache.

It has been shown that output dependences (write-after-write hazards) and antidependences (write-after-read hazards) can significantly reduce the available parallelism in a program [15]. Since these two types of dependences can be removed either at compile-time or at run-time using renaming techniques, they can be selectively ignored in the timing simulator. The final output of the simulator is the execution time for a shared memory multiprocessor with $p$ processors where each processor is capable of issuing $j$ instructions per cycle, and the execution stage of each functional unit is divided into $m$ pipeline segments.

## 4. Performance Results for Single-Granularity Processors

This section presents the simulation results for the multiprocessor, the pipelined processor, and the superscalar processor when exploiting only a single level of parallelism granularity. The first subsection measures the maximum inherent parallelism available in the application programs used in these experiments. Section 4.2 identifies several factors important in the design of the three types of architectures and examines the impact of these factors on the relative speedups of the individual machines.

### 4.1. Maximum Speedups

Relative speedup is used as the figure of merit in these simulations. The speedup for architectural configuration $x$ is defined to be $S_x=T_1/T_x$, where $T_1$ is the execution time for a given program on the basis processor, and $T_x$ is the execution time for the same program on configuration $x$. The maximum speedup of the test programs used in these experiments is found by simulating the execution of the programs on a superscalar processor with an unlimited number of functional units and perfect branch prediction. The resulting minimum execution time, $T_{\min}$, then determines the maximum possible speedup as $S_{\max}=T_1/T_{\min}$. Since Amdahl's Law limits the maximum speedup of a program to be $S_{\max}=1/\alpha$, where $\alpha$ is the dynamic fraction of code that must be executed sequentially [20], the application programs are categorized using $\alpha$, as shown in Table 2. All of the programs within a given category were found to have approximately the same performance characteristics which are well represented by the geometric mean of the category. Consequently, only the geometric means of the speedup values for each category are plotted in the following graphs.

While the dependences inherent in a program limit its maximum speedup to $S_{\max}$, its actual speedup on a particular machine also will be limited by the available parallelism in the machine. This parallelism limit is defined to be the *degree of architectural parallelism* of the given system configuration, denoted $D_{arch}$. This parameter provides an upper bound on the parallel speedup that can be obtained on the given machine due to resource limitations. For example, the maximum speedup of a program executing on a multiprocessor is the smaller of its inherent parallelism, $S_{\max}$, and the number of processors, $p$. Similarly, the maximum speedup is limited to $j$ in the superscalar processor, and to $m$ in the pipelined processor. Other factors, such as memory and synchronization delays, may additionally limit the actual speedup to be less than either $S_{\max}$ or $D_{arch}$. The machine models used in these simulations are summarized in Table 3 along with the corresponding values of $D_{arch}$.

Table 2: Maximum speedup values for the test programs.

| Category | | Program | Description | α | $S_{max}$ |
|---|---|---|---|---|---|
| 1. | $0<\alpha\leq0.03$ | k07 | Livermore Loops | 0.002 | 646 |
| | $(33.3\leq S_{max}<\infty)$ | k01 | Livermore Loops | 0.003 | 288 |
| | | k12 | Livermore Loops | 0.006 | 166 |
| | | k08 | Livermore Loops | 0.010 | 101 |
| 2. | $0.03<\alpha\leq0.12$ | elmbak | Eispack | 0.045 | 22.2 |
| | $(8.3\leq S_{max}<33.3)$ | balbak | Eispack | 0.066 | 15.2 |
| | | pcell | Particle-in-cell (16) | 0.118 | 8.5 |
| 3. | $0.12<\alpha\leq0.50$ | bisect | Eispack | 0.185 | 5.4 |
| | $(2.0\leq S_{max}<8.3)$ | bandr | Eispack | 0.249 | 4.0 |
| | | fft512 | Fast Fourier transform (512-pt.) | 0.263 | 3.8 |
| | | gss20 | Gaussian elimination (20x20) | 0.268 | 3.7 |
| | | tred1 | Eispack | 0.298 | 3.4 |
| | | simple | Hydrodynamics (8x8) | 0.304 | 3.3 |
| | | hqr2 | Eispack | 0.306 | 3.3 |
| | | hqr | Eispack | 0.338 | 3.0 |
| | | linpack | Linear system solver (20x20) | 0.407 | 2.5 |
| | | bmk1 | Integer benchmark | 0.494 | 2.0 |
| 4. | $0.50<\alpha\leq1.0$ | balanc | Eispack | 0.620 | 1.6 |
| | $(1.0\leq S_{max}<2.0)$ | imtql2 | Eispack | 0.620 | 1.6 |
| | | mem50 | Maximum entropy spectrum | 0.658 | 1.5 |

Table 3: Summary of machine models.
($l_i$ is the latency for op-code $i$)

| Machine type | #procs | instrs issued per cycle | cycle time | latency cycles | latency time | $D_{arch}$ |
|---|---|---|---|---|---|---|
| Basis | 1 | 1 | $T$ | $l_i$ | $Tl_i$ | 1 |
| Multiprocessor | $p$ | 1 | $T$ | $l_i$ | $Tl_i$ | $p$ |
| Superscalar | 1 | $j$ | $T$ | $l_i$ | $Tl_i$ | $j$ |
| Pipeline | 1 | 1 | $T/m$ | $ml_i$ | $Tl_i$ | $m$ |
| Combination | $p$ | $j$ | $T/m$ | $ml_i$ | $Tl_i$ | $pjm$ |

### 4.2. Performance Factors

There are a variety of factors that influence the performance of the individual parallel architectures in different ways. In particular, this section examines the performance impact of branch bypassing and register renaming on the superscalar and pipelined processors, the performance impact of maintaining cache coherence on the shared memory multiprocessor, and the performance impact on all three architectures of executing programs with limited inherent parallelism.

### 4.2.1. Branch Bypassing

In a multiprocessor, iterations from a parallel loop with no interiteration dependences are executed in parallel by distributing the iterations across all of the processors, as shown in Figure 2. The pipelined and superscalar processors, on the other hand, must *bypass* branches to look-ahead into subsequent iterations to find independent operations to issue concurrently. The number of branches bypassed is defined to be the number of loop iterations that are unrolled, or the number of successive branches that are predicted correctly. Figure 3 shows the speedup of a pipelined processor with $m$=32 and a superscalar processor with $j$=32 as the number of branches dynamically bypassed is varied when executing the very parallel Category 1 programs. This figure shows that to match the performance of the multiprocessor with $p$=32, the fine-grained processors must bypass at least 128

| Multiprocessor | | | | Fine-Grained processor |
|---|---|---|---|---|
| Proc 0 | Proc 1 | ... | Proc $p-1$ | |
| $S_1(1)$ | $S_1(2)$ | ... | $S_1(p)$ | $S_1(1)$ |
| ... | ... | ... | ... | ... |
| $S_k(1)$ | $S_k(2)$ | ... | $S_k(p)$ | $S_k(1)$ |
| $S_1(p+1)$ | $S_1(p+2)$ | ... | $S_1(2p)$ | $S_1(2)$ |
| ... | ... | ... | ... | ... |
| $S_k(p+1)$ | $S_k(p+2)$ | ... | $S_k(2p)$ | $S_k(2)$ |
| $S_1(2p+1)$ | $S_1(2p+2)$ | ... | $S_1(3p)$ | $S_1(3)$ |
| ... | ... | ... | ... | ... |
| $S_k(2p+1)$ | $S_k(2p+2)$ | ... | $S_k(3p)$ | $S_k(3)$ |
| ... | ... | ... | ... | ... |

Figure 2: Differences in look-ahead order.

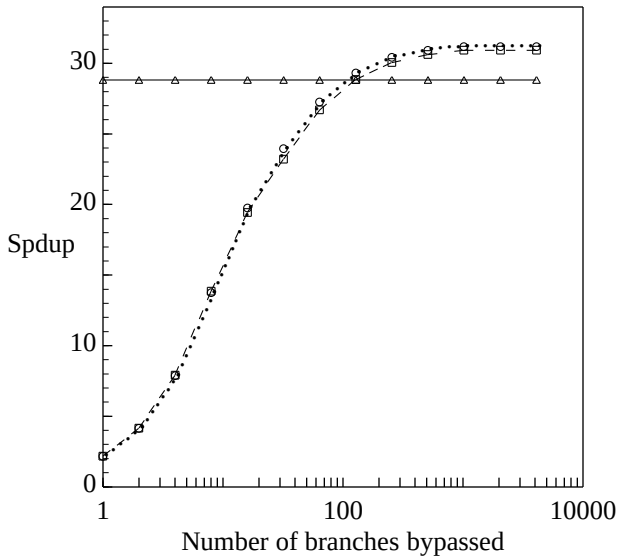


Figure 3: Effect of branch bypassing with the Category 1 programs.
(solid=multiprocessor; dash=superscalar; dot=pipelined)

branches, meaning that loops must be unrolled 128 times, or branches must be predicted with an accuracy greater than 99 percent. As more than 128 branches are bypassed, the performance of the superscalar and the pipelined processors exceeds that of the multiprocessor since there is additional fine-grained parallelism exploited by these processors that is not exploited by this coarse-grained multiprocessor.

### 4.2.2. Register Allocation

In addition to the need for branch bypassing, all three architectures must intelligently allocate registers to prevent the introduction of ''false dependences'' that will limit the available parallelism. To execute operations from different iterations concurrently, each iteration must be treated as a separate context. For example, a loop iteration may include something like the following two assembly level operations:

| | | |
|---|---|---|
| $S_1$: | load_index | index_reg,base_addr,r1 |
| $S_2$: | add | r1,r0,r1 |

A different memory location will be loaded into r1 for each iteration of the loop since the index register will have a different value in each iteration. A processor using dynamic dependence checking will detect a flow dependence from statement $S_2$ in iteration $i$ to statement $S_2$ in iteration $i$+1, thereby preventing the two operations from executing simultaneously. This false data dependence is detected even though at the source code level, no such dependence exists.

The separate, independent register files in each processor of the multiprocessor automatically generate unique contexts for each iteration. Thus, in the multiprocessor, the compiler needs to allocate registers only for a single iteration. Since subsequent iterations execute on physically separate processors, the same register name will reference a different physical register. The fine-grained processors, however, must use a register renaming mechanism to generate the separate contexts for each iteration. This renaming can done either statically by the compiler [8, 18], or dynamically by the hardware at run-time [27, 32, 34].

Figure 4 quantifies the effect of this register renaming requirement on the performance of the superscalar processor when executing the very parallel Category 1 programs. To show the effects of register renaming only, these simulations assume perfect branch prediction with all memory references satisfied by the cache. With a small degree of architectural parallelism, the few available functional units in the processor are kept busy with operations from only a few iterations. As a result, any false dependences introduced by the lack of register renaming produce very little performance impact. As the architectural parallelism is increased, however, these false dependences reduce the programs' exploitable parallelism which thereby severely restricts the processor's performance compared to its performance with an adequate register renaming mechanism. These simulations suggest that there is an adequate amount of parallelism available in the programs to utilize a processor with $D_{arch} \simeq 4$ without register renaming. Thus, a register renaming mechanism is needed only when both $D_{arch}$ and the inherent parallelism in the program are greater than approximately four. Similar results were found for the pipelined processor, also.

### 4.2.3. Multiprocessor Cache Coherence

A significant disadvantage of the multiprocessor is the complex memory system required to maintain coherent access to shared variables when using private data caches. Figure 5 compares the performance of the multiprocessor with an infinite data cache when using an ideal, zero overhead coherence mechanism to the performance obtained when using a central directory mechanism [4] to

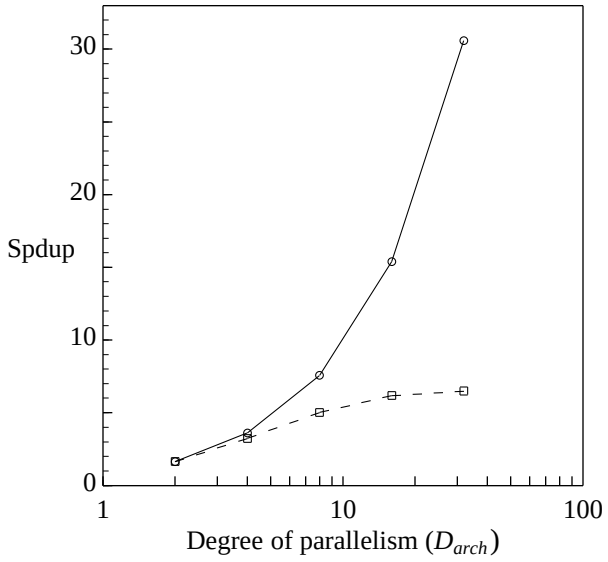


Figure 4: The performance impact of register renaming on the superscalar processor.
solid = with register renaming
dash = without register renaming

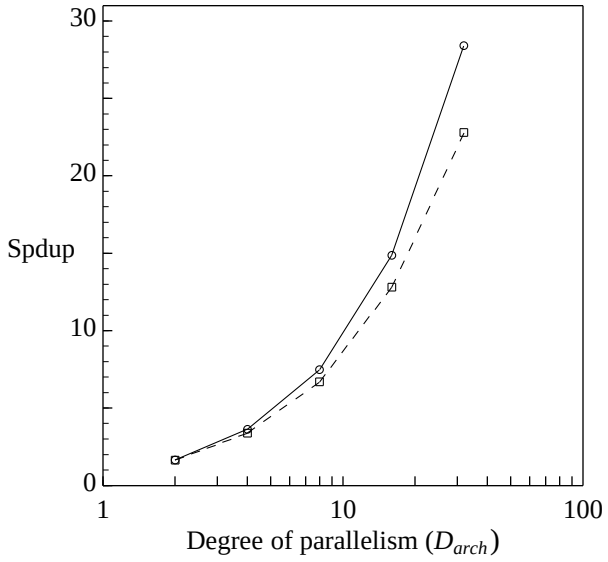


Figure 5: The performance impact of the cache coherence penalty in the multiprocessor.
solid = ideal (no overhead) coherence enforcement
dash = directory-based coherence enforcement

maintain coherence. With few processors (i.e. small $D_{arch}$), memory delays due to coherence enforcement are relatively small compared to the total execution time. Thus, coherence enforcement has little performance impact for these small systems. As the number of processors increases, however, there is more sharing of data, which then produces more invalidation traffic and more

memory traffic. The combination of increased network traffic and reduced hit ratios significantly increases the average time required to access memory, which adds directly to the total execution time. Since the fine-grained processors exploit parallelism using only a single processor, all of the memory accesses from all of the functional units use a single data cache. As a result, these fine-grained parallel processors avoid the coherence problem.

### 4.2.4. Impact of Limited Program Parallelism

The previous simulations demonstrated the performance impact of various architectural factors for application programs with an inherent parallelism greater than the architectural parallelism of the system. In contrast, Figure 6 shows the relative speedups for the three different architectures when executing the moderately parallel Category 2 programs. With limited architectural parallelism (small $D_{arch}$), all three configurations have approximately the same ability to exploit parallelism. Both the superscalar processor and the pipelined processor perform slightly better than the multiprocessor due to the multiprocessor's synchronization overhead. As $D_{arch}$ increases, the superscalar processor encounters greater memory delays than the pipelined processor which decreases the performance of the superscalar processor relative to the pipelined processor. The multiprocessor achieves greater speedup than either the superscalar or the pipelined processors with increasing $D_{arch}$ because these fine-grained processors cannot look-ahead far enough in the instruction stream for parallelism opportunities when executing the parallel loops.

The peak in the speedup curve for the multiprocessor in Figure 6 suggests that when executing these moderately parallel programs, it is important to limit the number of processors so that the increasing memory and synchronization delays do not negate the benefits of using the additional processors. A hierarchical structure similar to the Cedar system [14] may be the best architectural approach since programs (or program sections) with smaller inherent parallelism can be assigned to a single cluster of processors to minimize memory and synchronization delays. Because most memory

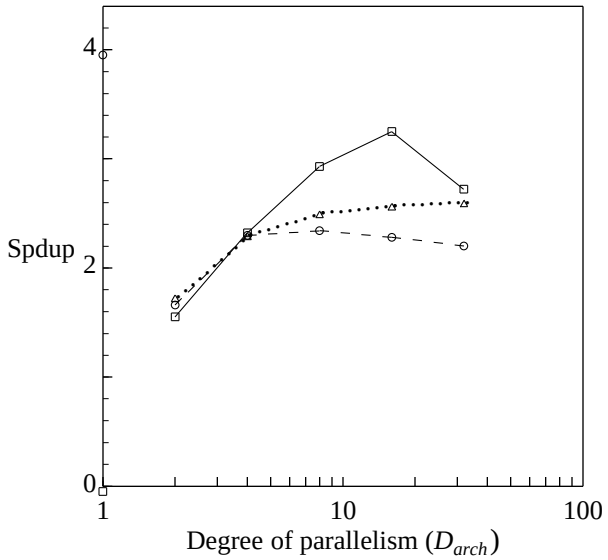


Figure 6: Speedups of Category 2 programs.
(solid=multiprocessor; dash=superscalar; dot=pipelined)

accesses can be made to the local cluster memory, the average memory delay will be reduced and there will be little network interference with the other clusters.

The speedups of the programs with relatively low levels of inherent parallelism are shown in Figures 7 and 8. In both of these program categories, the pipelined processor performs the best due to the logarithmically increasing memory delays in the superscalar processor as $D_{arch}$ increases, and due

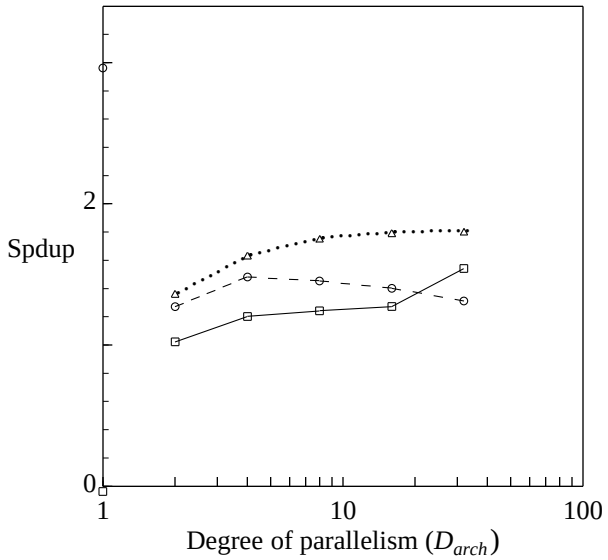


Figure 7: Speedups of Category 3 programs.
(solid=multiprocessor; dash=superscalar; dot=pipelined)

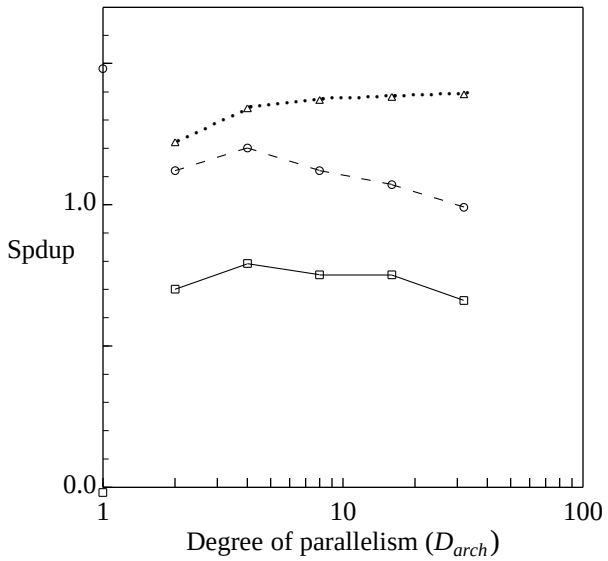


Figure 8: Speedups of Category 4 programs.
(solid=multiprocessor; dash=superscalar; dot=pipelined)

to the increasing memory and synchronization delays in the multiprocessor. Because of the small inherent parallelism in these programs, the architectural differences between the processors produce a relatively small difference in the realized speedups. The performance differences of the three architectures are primarily due to their differences in memory delays, and due to the synchronization requirements in the multiprocessor. For the very sequential programs in Figure 8, the multiprocessor actually has lower performance than the basis processor because of its synchronization overhead. These figures indicate that the fine-grained processors have the best ability to extract what little parallelism is available in these more sequential programs.

## 5. A Mixed Granularity Architecture

Not surprisingly, the previous simulations suggest that for sections of a program with low inherent parallelism, the pipelined and superscalar processors will produce the greatest speedups compared to the multiprocessor due to their low communication costs. For program sections with high inherent parallelism, however, the multiprocessor's relatively high memory and synchronization delays can be masked by simultaneously executing the independent iterations of a parallel loop on its individual processors. The superscalar and pipelined processors have a more difficult time looking ahead into the dynamic instruction stream in these very parallel loops and so produce lower performance than the multiprocessor. These simulation results thus suggest that an architecture that combines the features of both the fine-grained and the coarse-grained parallel processors should be able to produce better performance than any of the three architectures by themselves.

To test this idea, the fine-grained and coarse-grained architectures are combined in a shared memory multiprocessor with $p$ processors, where each processor is either pipelined to degree $m$, or each is a superscalar processor capable of issuing $j$ instructions per cycle. The total degree of architectural parallelism for the pipelined multiprocessor is $D_{arch}=pm$, and is $D_{arch}=pj$ for the superscalar multiprocessor. To find the best mix of fine-grained and coarse-grained parallelism for these combined architectures, it is necessary to determine what combinations of $p$, $j$, and $m$ produce the greatest speedup. In the following simulations, the total degree of architectural parallelism, $D_{arch}$, is held constant while the degree of fine-grained parallelism within each processor, and the total number of processors, is changed. Thus, with $m$=1 in the pipelined multiprocessor, or $j$=1 in the superscalar multiprocessor, the number of processors is $p=D_{arch}=c$, where $c$ is some constant. This configuration then is a completely coarse-grained multiprocessor. As $m$ and $j$ are increased, the number of processors in the two systems is reduced to $p=c/m$ and $p=c/j$ until a completely fine-grained architecture is simulated when $p$=1, and $m=c$ and $j=c$.

Figure 9 uses this simulation strategy to show the speedups for each of the four program categories when executed on the pipelined multiprocessor and the superscalar multiprocessor with the total degree of architectural parallelism fixed at $D_{arch}=c$=32. Points further to the left in this figure are configurations that use more fine-grained parallelism while further to the right, the individual processors have less fine-grained parallelism, but more of them are used. For example, at the point where 8 processors are used, each processor is pipelined to a degree of 4, or each is capable of issuing up to 4 instructions per cycle. When the number of processors is doubled to 16, the degree of fine-grained parallelism in each of the processors is reduced to 2 thereby maintaining the total architectural parallelism at 32.

The initially positive slope of the top curve in Figure 9 shows that it is generally better to exploit coarse-grained parallelism using a multiprocessor when executing the very parallel programs in Category 1. The maximum at 16 processors, however, indicates that there is some fine-grained parallelism within each loop iteration, as well as in the ''serial'' code between parallel loops, that is

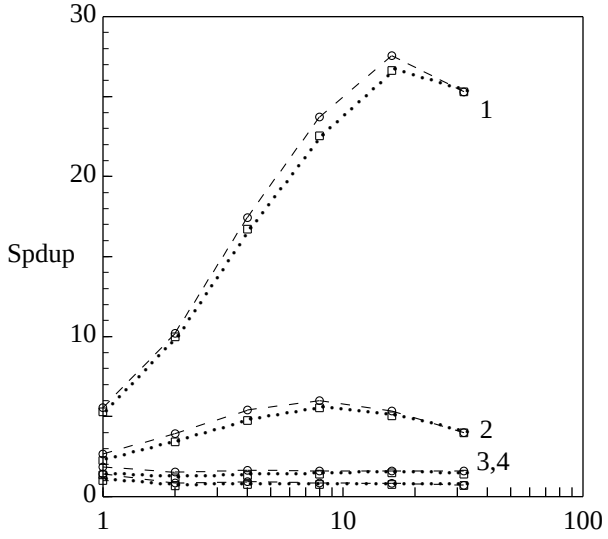


Figure 9: Combined architecture with $D_{arch}$=32.
1) $0<\alpha\leq 0.03$;
2) $0.03<\alpha\leq 0.12$;
3) $0.12<\alpha\leq 0.50$;
4) $0.50<\alpha\leq 1.0$.
(dot=superscalar multiprocessor; dash=pipelined multiprocessor)

not being exploited by the multiprocessor alone. Thus, a combined fine-grained/coarse-grained architecture is required to exploit the maximum parallelism inherent in the programs. A similar peak in the speedup curve for the Category 2 programs occurs when 8 processors are used, indicating that there is more fine-grained parallelism in these programs than in the very parallel programs. This peak again suggests that the combined architecture is necessary to maximize performance. While there is not much parallelism available in the programs in Categories 3 and 4, there is still a small amount of fine-grained parallelism that can be exploited by the individual pipelined and superscalar processors of the multiprocessor. This small amount of parallelism is not very well exploited by the completely coarse-grained multiprocessor, however.

To determine if the amount of fine-grained parallelism that should be used in each of the processors when executing the two most parallel program categories is constant, or whether it varies as the total architectural parallelism is changed, the pipelined multiprocessor and the superscalar multiprocessor were simulated for these two program categories with different values of $D_{arch}$. Figure 10 shows the speedups for the highly parallel Category 1 programs for values of $D_{arch}$ ranging from 8 to 64. For each curve, the total degree of architectural parallelism is held constant at the indicated value of $D_{arch}$ while the mix of fine-grained and coarse-grained parallelism is varied along the horizontal axis. The maximum speedup for $D_{arch}$=64 occurs when using 32 processors, each with a fine-grained parallelism of 2, or when using 64 processors that have no fine-grained parallelism capabilities. For $D_{arch}$=32, the best performance is obtained when using a fine-grained parallelism of 2 in each of 16 processors. Similarly, the best performance for $D_{arch}$=16 and $D_{arch}$=8 occurs when

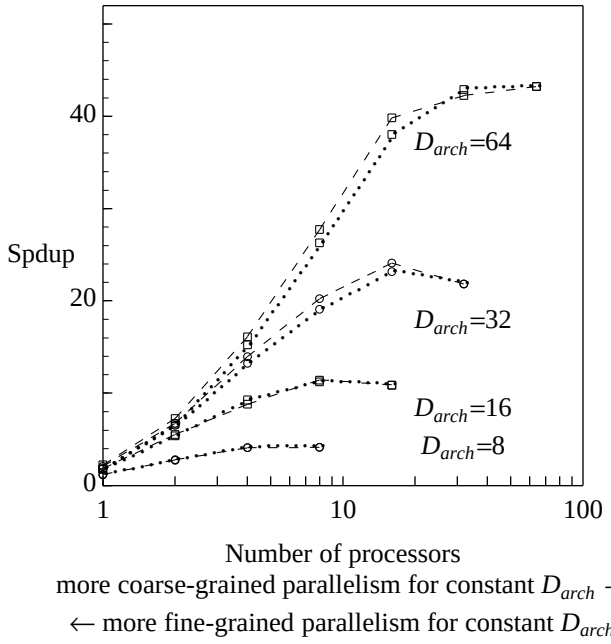


Figure 10: Effect of changing $D_{arch}$ in the combined architecture for the Category 1 programs. (dot=superscalar multiprocessor; dash=pipelined multiprocessor)

using $p$=8 and $p$=4 processors, respectively, where each processor has a fine-grained parallelism of 2. Thus, for these Category 1 test programs when using processors with the characteristics described in Section 3, the maximum performance tends to occur when $D_{arch}=2p$. That is, the maximum performance is obtained by using a multiprocessor configuration built of processors with a fine-grained parallelism capability of two, independent of the number of processors that are used.

Figure 11 repeats these simulations for the less parallel programs in Category 2. For these programs, the best performance occurs when $D_{arch}=4p$; that is, when each of the processors has a fine-grained parallelism capability of about four, independent of the number of processors used. Although it is not shown in these figures, additional simulations of a pipelined superscalar multiprocessor showed that the maximum performance occurred when $D_{arch}=ap$ where the total fine-grained parallelism in each processor, $a=jm$, was two to four.

The simulations in both Figures 10 and 11 assumed that the network delay increased logarithmically with the number of processors. The specific network delay in these simulations provided a memory access time of $T_{read}=17+\lceil 1.5\log_2 p\rceil$ cycles. In fact, for large machines, or for machines using processors with very fast clock rates, the network delay may actually increase proportional to $p^{1/2}$ or $p^{1/3}$ due to the long network delay relative to the clock rate. To measure the effect of a slower network, Figures 12 and 13 repeat the simulations of Figures 10 and 11 when using a network that gives a memory delay of $T_{read}=17+\lceil 2p^{1/2}\rceil$ cycles. In these figures, the maximum speedup values are reduced slightly compared to the previous simulations due to the longer network delays. The peaks in the curves, however, still occur when each of the processors can exploit a fine-grained parallelism of 2 to 4 regardless of how many processors are used. Thus, these simulations suggest that for the given test programs, the individual processors in a multiprocessor system should be capable of exploiting a degree of fine-grained parallelism of two to four using any combination of

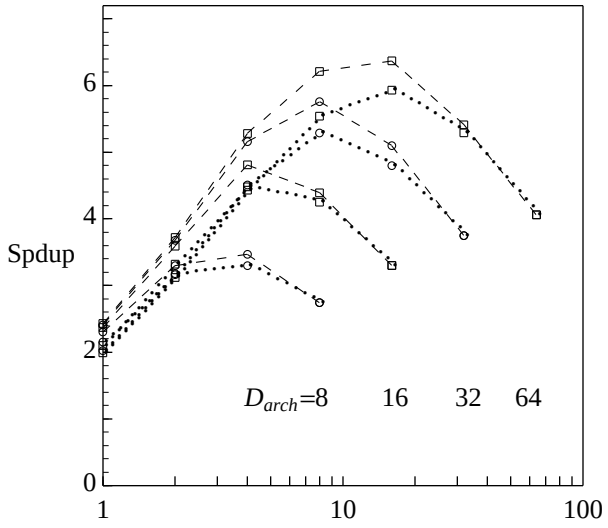


Figure 11: Effect of changing $D_{arch}$ in the combined architecture for the Category 2 programs. (dot=superscalar multiprocessor; dash=pipelined multiprocessor)

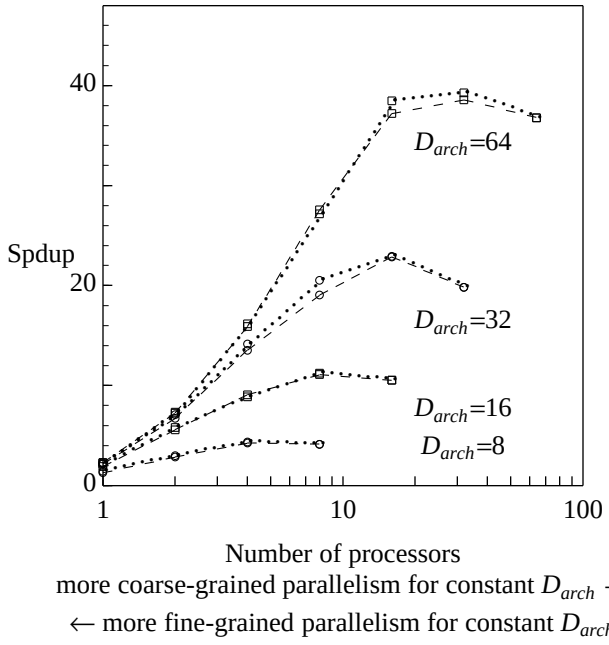


Figure 12: Effect of changing $D_{arch}$ in the combined architecture with a slow interconnection network for the Category 1 programs. (dot=superscalar multiprocessor; dash=pipelined multiprocessor)

pipelining and multiple instruction issuing. Since the simulations with the slower interconnection network made the peaks in the speedup curves slightly more pronounced than with the faster network, it is reasonable to expect that a similar mix of granularities would produce the best performance in systems with more processors than the moderate number of processors (8 to 64) used in this study. However, care must be taken when trying to extend these conclusions to larger systems due to the nonlinear increases in synchronization and cache coherence costs that would be incurred as the system size is increased.

## 6. Conclusions

This paper has presented a simulation methodology for examining the performance trade-offs of fine-grained and coarse-grained parallel processor architectures. Using computation-intensive numeric application programs with a wide range of inherent parallelism, these simulations have confirmed previous studies that have concluded that pipelined processors and superscalar processors using dynamic dependence checking are roughly equivalent in their abilities to exploit fine-grained parallelism. These simulations also showed that the two fine-grained processors outperformed a multiprocessor architecture when executing programs with low inherent parallelism due to their small average memory delays compared to the multiprocessor, and due to their nonexistent synchronization costs. When executing very parallel loops, however, these fine-grained processors must bypass an unrealistically large number of branches to match the performance of the coarse-grained parallelism of the shared memory multiprocessor. In addition, using the high-level structure identified by a parallelizing compiler can simplify the register allocation problem in the multiprocessor compared to the fine-grained processors. The primary disadvantage of the multiprocessor is the complex memory needed to maintain coherence among the private data caches, and the consequent reduction in

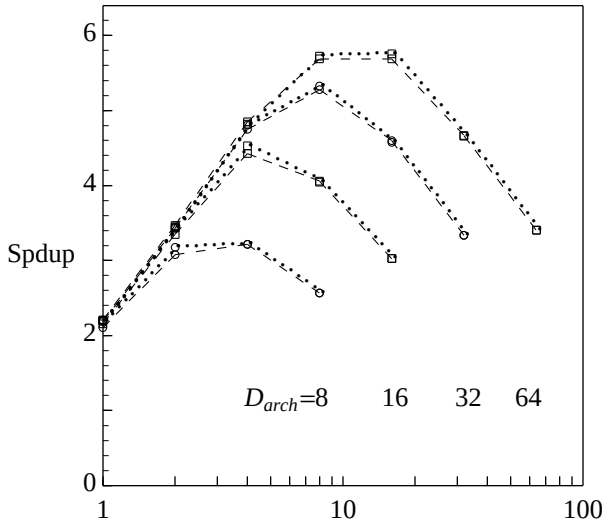


Figure 13: Effect of changing $D_{arch}$ in the combined architecture with a slow interconnection network for the Category 2 programs. (dot=superscalar multiprocessor; dash=pipelined multiprocessor)

performance.

To exploit the advantages of both the fine-grained and the coarse-grained architectures, two combined ‘‘multigrained’’ architectures were evaluated. Both architectures were shared memory multiprocessors with $p$ processors, but in one case the individual processors were pipelined to degree $m$, while in the other case, each processor of the multiprocessor was a superscalar processor capable of issuing up to $j$ instructions per cycle. The number of processors and the degree of fine-grained parallelism within each processor was varied while keeping the total degree of architectural parallelism of the entire system constant at $D_{arch}=c=pm$ in the pipelined multiprocessor and $D_{arch}=c=pj$ in the superscalar multiprocessor. This simulation technique was used to find the best mix of fine-grained and coarse-grained parallelism.

It was found that the best performance when executing these test programs occurs when any number of processors are used, but each of the individual processors have a fine-grained parallelism capability of two to four. (These simulations were performed for a range of $D_{arch}$=8 to 64 using processors with the characteristics described in Section 3, so care must be taken when extending these conclusions to larger systems or to processors with significantly different performance characteristics.) The multiprocessor configuration allows the simple exploitation of the high-level structure of the parallel loop iterations, while the pipelining or multiple instruction issuing of the individual processors can exploit the relatively small amount of fine-grained parallelism between and within the parallel loops. Furthermore, these simulations suggest that program sections with limited inherent parallelism can be scheduled on a single cluster of a hierarchical configuration to exploit the available parallelism without the memory delays encountered when using all of the processors. In addition, the separate program counters in the multiprocessor add a flexibility to this combined architecture that is beyond the capability of the fine-grained processors alone. Thus, these

simulations suggest that it is better to construct a system using a large number of slightly parallel processors rather than using only a few processors each with a high degree of fine-grained parallelism.

**Acknowledgements**

Thanks to Carl Beckmann for his insights into multiprocessor synchronization, and to John Andrews and Pen-Chung Yew for their helpful comments on an early version of this paper. A preliminary version of this work was presented at the Twenty-Fourth Annual Hawaii International Conference on System Sciences [19]. Portions of this work were performed at the University of Illinois at Urbana-Champaign while the author was supported by the National Science Foundation under Grant No. MIP-8410110, with additional support from NASA Ames Research Center Grant No. NCC 2-559 (DARPA), National Science Foundation Grant No. MIP-88-07775, and Department of Energy Grant No. DE-FG02-85ER25001.

**References**

1. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, ‘‘The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling,’’ *IBM Journal of Research and Development*, Vol. 11, No. 1, pp. 8-24, January 1967.

2. Arvind, David E. Culler, and Gino K. Maa, ‘‘Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs,’’ *Proceedings of Supercomputing ’88*, pp. 60-69, November 1988.

3. M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, ‘‘Single Instruction Stream Parallelism is Greater than Two,’’ *International Symposium on Computer Architecture*, pp. 276-286, 1991.

4. Lucien M. Censier and Paul Feautrier, ‘‘A New Solution to Coherence Problems in Multicache Systems,’’ *IEEE Transactions on Computers*, Vol. C-27, No. 12, pp. 1112-1118, December 1978.

5. Robert P. Colwell, Robert P. Nix, John J. O’Donnell, David B. Papworth, and Paul K. Rodman, ‘‘A VLIW Architecture for a Trace Scheduling Compiler,’’ *IEEE Transactions on Computers*, Vol. 37, No. 8, pp. 967-979, August 1988.

6. Ron Cytron, ‘‘Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract),’’ *International Conference on Parallel Processing*, pp. 836-844, 1986.

7. Philip G. Emma and Edward S. Davidson, ‘‘Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance,’’ *IEEE Transactions on Computers*, Vol. C-36, No. 7, pp. 859-875, July 1987.

8. Joseph A. Fisher, ‘‘Trace Scheduling: A Technique for Global Microcode Compaction,’’ *IEEE Transactions on Computers*, Vol. C-30, No. 7, pp. 478-490, July 1981.

9. Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, ‘‘The NYU Ultracomputer -- Designing a MIMD, Shared-Memory Parallel Machine,’’ *International Symposium on Computer Architecture*, pp. 27-42, 1982.

10. Norman P. Jouppi, ‘‘The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance,’’ *IEEE Transactions on Computers*, Vol. 38, No. 12, pp. 1645-1658, December 1989.

11. Norman P. Jouppi, ‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU,’’ *International Symposium on Computer Architecture*, pp. 281-289, May 1989.

12. Norman P. Jouppi and David W. Wall, ‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272-282, April 1989.

13. Clyde P. Kruskal and Marc Snir, ‘‘The Performance of Multistage Interconnection Networks for Multiprocessors,’’ *IEEE Transactions on Computers*, Vol. C-32, No. 12, pp. 1091-1098, December 1983.

14. David J. Kuck, Edward S. Davidson, Duncan J. Lawrie, and Ahmed H. Sameh, ‘‘Parallel Supercomputing Today and the Cedar Approach,’’ *Science*, Vol. 231, pp. 967-974, 28 February 1986.

15. Manoj Kumar, ‘‘Effect of Storage Allocation/Reclamation Methods on Parallelism and Storage Requirements,’’ *International Symposium on Computer Architecture*, pp. 197-205, June 1987.

16. Manoj Kumar, ‘‘Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications,’’ *IEEE Transactions on Computers*, Vol. 37, No. 9, pp. 1088-1098, September 1988.

17. Steven R. Kunkel and James E. Smith, ‘‘Optimal Pipelining in Supercomputers,’’ *International Symposium on Computer Architecture*, pp. 404-411, 1986.

18. Monica Lam, ‘‘Software Pipelining: An Effective Scheduling Technique for VLIW Machines,’’ *Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation*, pp. 318-328, June 1988.

19. David J. Lilja and Pen-Chung Yew, ‘‘The Performance Potential of Fine-Grain and Coarse-Grain Parallel Architectures,’’ *Hawaii International Conference on System Sciences, Vol. I: Architecture*, pp. 324-333, 1991.

20. David J. Lilja (ed.), *Architectural Alternatives for Exploiting Parallelism*, IEEE Computer Society Press, Los Alamitos, CA, ISBN 0-8186-2642-9, 1992.

21. Kazuaki Murakami, Naohiko Irie, Morihiro Kuga, and Shinji Tomita, ‘‘SIMP (Single Instruction stream/Multiple instruction Pipelining): A Novel High-Speed Single-Processor Architecture,’’ *International Symposium on Computer Architecture*, pp. 78-85, May 1989.

22. Alexandru Nicolau and Joseph A. Fisher, ‘‘Measuring the Parallelism Available for Very Long Instruction Word Architectures,’’ *IEEE Transactions on Computers*, Vol. C-33, No. 11, pp. 968-976, November 1984.

23. Janak H. Patel, ‘‘Performance of Processor-Memory Interconnections for Multiprocessors,’’ *IEEE Transactions on Computers*, Vol. C-30, pp. 771-780, October 1981.

24. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, ‘‘The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture,’’ *International Conference on Parallel Processing*, pp. 764-771, 1985.

25. A. R. Pleszkun and G. S. Sohi, ‘‘The Performance Potential of Multiple Functional Unit Processors,’’ *International Symposium on Computer Architecture*, pp. 37-44, 1988.

26. George Radin, ‘‘The 801 Minicomputer,’’ *IBM Journal of Research and Development*, Vol. 27, No. 3, pp. 237-246, May 1983.

27. B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle, ‘‘The Cydra 5 Departmental Supercomputer,’’ *Computer*, Vol. 22, No. 1, pp. 12-35, January 1989.

28. Edward M. Riseman and Caxton C. Foster, ‘‘The Inhibition of Potential Parallelism by Conditional Jumps,’’ *IEEE Transactions on Computers*, Vol. C-21, No. 12, pp. 1405-1411, December 1972.

29. James E. Smith, ‘‘Dynamic Instruction Scheduling and the Astronautics ZS-1,’’ *Computer*, Vol. 22, No. 7, pp. 21-35, July 1989.

30. Michael D. Smith, Mike Johnson, and Mark A. Horowitz, ‘‘Limits on Multiple Instruction Issue,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290-302, April 1989.

31. Gurindar S. Sohi and Sriram Vajapeyam, ‘‘Tradeoffs in Instruction Format Design for Horizontal Architectures,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 15-25, April 1989.

32. C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. P. Shen, ‘‘Instruction Level Profiling and Evaluation of the IBM RS/6000,’’ *International Symposium on Computer Architecture*, pp. 180-189, 1991.

33. Garold S. Tjaden and Michael J. Flynn, ‘‘Detection and Parallel Execution of Independent Instructions,’’ *IEEE Transactions on Computers*, Vol. C-19, No. 10, pp. 889-895, October 1970.

34. R. M. Tomasulo, ‘‘An Efficient Algorithm for Exploiting Multiple Arithmetic Units,’’ *IBM Journal of Research and Development*, Vol. 11, No. 1, pp. 25-33, January 1967.

35. David W. Wall, ‘‘Limits of Instruction-Level Parallelism,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, 1991.

36. Shlomo Weiss and James E. Smith, ‘‘Instruction Issue Logic in Pipelined Supercomputers,’’ *IEEE Transactions on Computers*, Vol. C-33, No. 11, pp. 1013-1022, November 1984.