Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator

Wen-mei W. Hwu
Pohua P. Chang

Coordinated Science Laboratory
1101 W. Springfield Ave.
University of Illinois
Urbana, IL 61801

ABSTRACT

With advances in VLSI technology, microprocessor designers can provide more microarchitectural parallelism to increase performance. We have identified four major forms of such parallelism: multiple microoperations issued per cycle, multiple result distribution buses, multiple execution units, and pipelined execution units. The experiments reported in this paper address two important issues: the effects of these forms and the appropriate balance among them. A *central* microarchitecture is identified as the comparison basis. We separately vary each form of the microarchitectural parallelism in the *central* to measure their individual effects on performance. In addition, we vary two forms of the microarchitectural parallelism in the *central* to derive an appropriate balance between them. To make fair comparisons, our compiler generates different code sequences optimized for different microarchitectural configurations. For each given set of technology constraints, these experiments can be used to derive a cost-effective microarchitecture to execute each given set of workload programs at high speed.

1. Introduction

The demand for high speed microprocessors continues to increase. Applications include high performance workstations, application specific processors, and implementations of mini and mainframe computers. The design constraints of these microprocessors are, however, different enough from those of the conventional technologies to force the researchers to pursue new design approaches to achieve high performance. One of the most promising approaches to designing high speed microprocessors is to use a sophisticated compiler to exploit the parallelism both in the programs and in the microarchitecture. The compiler and the parallel microarchitecture work closely together to achieve high performance without incurring excessive hardware cost.

On the microarchitecture side, concurrency can be provided in the form of parallel data paths and pipelining. These two forms together support fetching, issuing, executing, and distributing results for multiple microoperations per cycle. The experiments reported in this paper are based on a class of microarchitectures with pipelining and parallel data paths. All these microarchitectures rely on the compiler to directly generate microprograms; there is no intermediate level of instruction set architecture.

On the compiler side, source level transformations are first performed to increase the program parallelism visible to the code generator. Register allocation cooperates with the code scheduling to enhance the effectiveness of the latter [1]. By working closely with code scheduling, register allocation can use just enough registers to support the desired parallel execution.

In this paper, we report several experiments on using a versatile optimizing compiler to evaluate the benefit of four forms of microarchitectural parallelism: multiple microoperations issued per cycle, multiple result distribution buses, multiple execution units, and pipelined execution units. We use the first 14 of the Livermore loops [2] and 10 of the linpack subroutines [3] as our preliminary benchmarks. The compiler generates optimized code for different microarchitecture configurations. We show how the compiler can help to derive a balanced design for high performance. Ultimately, we would like to automatically generate a well balanced microarchitecture given a set of workloads written in C and a set of technology constraints.

This paper is organized into four sections. Section 2 gives some background regarding our research. Section 3 reports and analyzes our experimental results. Section 4 offers some concluding remarks.

2. Background

In this section, we provide some background information regarding the optimization code generation techniques, the parallel microarchitectures, the benchmarks, and some previous works.

2.1. The Optimization and Code Generation Techniques

Several source level transformation techniques, including loop unrolling, loop folding, and code migration are applied to the benchmarks before code generation. Loop unrolling generates a loop header, for an inner loop, to ensure that the number of remaining iterations is a multiple of N (N>1), and then generates a N-unrolled version of the loop. The resultant loop body forms a large basic block. Code migration moves the bottom portions of long data dependency links to the

This research has been partly supported by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), a NASA-supported Center for Excellence.

destination basic blocks provided that the migrated portions do not affect the branch direction. Loop folding is a special case of code migration for inner loops. Code duplication and variable renaming are used to support code migration [1, 4].

The code generator receives an intermediate form which consists of directed control graphs: the nodes are basic blocks and the arcs represent control transfers. Each basic block contains a list of three address codes which operate on the variables and structures defined in the source high level language (C in this case). The three address codes are translated into the microoperations which operate on the general purpose registers and memory locations. The microoperations are scheduled into program slots according to the data dependencies and the resource conflicts. The code generator works on one procedure at a time*. The functionality of the code generator is performed in seven steps: sorting basic blocks, selecting register variables, mapping variables into virtual registers, analyzing register lifetime, preliminary code reordering, mapping virtual registers into physical registers, and code scheduling [1,4,5,6]. We briefly describe these seven steps as follows.

Step 1: Sorting basic blocks. We sort the basic blocks according to their estimated execution frequencies. At present, our code generator examines the loop structure to estimate the execution frequency of each basic block. We are implementing an interface to a profile program so that the programmer can optionally use the profiling results to increase the accuracy of the estimation.

Step 2: Selecting register variables. We select the register variables according to their estimated dynamic access frequencies. At present, the register variables will stay in the register through the lifetime of the procedure. The dynamic access frequencies of variables are estimated by examining their access frequencies in each basic block and the estimated execution frequency of each basic block.

Step 3: Mapping variables into virtual registers. We map the variables into the infinitely many virtual registers. If a static variable or a parameter is selected as a register variable, we add *load* and *store* microoperations to the procedure entrance and/or exit to transfer the variable between the corresponding memory location and the corresponding virtual register. If a variable is not selected as a register variable, we add *load* and *store* microoperations within basic blocks to transfer variables between the corresponding memory location and the corresponding virtual register.

Step 4: For each basic block, we determine the *in*, *out*, *use*, and *define* sets of virtual registers [1]. The *in* set consists of the virtual registers whose contents are valid when control is transferred into the basic block. The *out* set consists of the virtual registers whose contents are valid when control is transferred out of the basic block. The *use* set consists of the virtual registers used before they are defined the basic block. The *define* set consists of the vritual registers serving as destination operands in the basic block. These sets together give us the register lifetime and flow information among basic blocks.

*We are implementing a set of optimization techniques based on interprocedural analysis.

Steps 5 through 7 are performed to one basic block at a time, from the basic block with the highest expected execution frequency to the one with the lowest. With this ordering, we can force the delay and spilling into less important basic blocks without facing the convergence problem. Code migration is only done from more important basic blocks to less important ones.

Step 5: Preliminary code reordering. Within each basic block, we reorder the three address codes according to the data dependencies. Mapping virtual registers to physical registers can introduce extra dependencies by recycling physical registers and inserting spill codes (see below). Without preliminary code reordering, the extra dependencies may prevent the proper code movement in the final scheduling. We characterize the effectiveness of this step in section 3.1.

Step 6: We map the infinite virtual registers into the finite physical registers using the global and local register lifetime information. Physical registers are recycled, introducing extra register dependencies. Spill codes may be inserted, introducing extra memory and register dependencies.

Step 7: We schedule the microoperations into the microinstructions of the target machine according to 1) the resource constraints and 2) the data dependencies, both from inside the basic block and from the more important basic blocks.

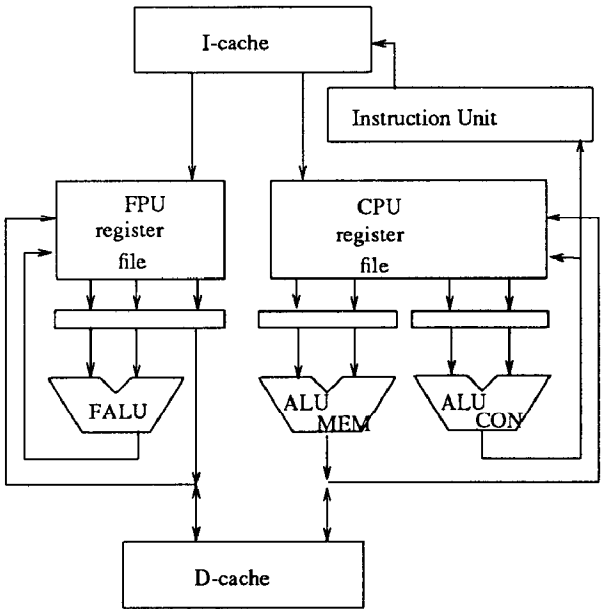

Figure 1. The *central* microarchitecture

2.2. The Parallel Microarchitectures

The microarchitectures we are designing provide parallelism through pipelining and parallel data paths [7, 8, 9]. There is no hardware interlocking or hardware data dependency resolution mechanism. The code scheduling phase of the code generator must handle all the data dependencies and resource conflicts. An example of this class of microarchitectures is the MIPS microprocessor [10]. We would like to determine a cost effective approach to improve performance by adding more parallelism into the microarchitecture.

The microarchitecture design parameters we have examined are (1) the number of microoperations issued per cycle, (2) the number of result distribution buses, (3) the execution unit pipeline structure, and (4) the number of execution units. Arbitrarily varying the design parameters creates too many design points in this 4-dimensional design space. Therefore, we first identify a *central* microarchitecture which can be efficiently implemented with the state-of-the-art fabrication technology; then we explore the neighborhood of the *central* by slightly varying the design parameters.

The *central* microarchitecture (see Figure 1) has (1) two microoperations issued per cycle, (2) two result distribution buses on both the CPU and the FPU, (3) one cache access port, one floating-point adder, and one floating-point multiplier with pipeline latencies 2, 4, and 6 cycles, respectively. The microarchitecture is equipped with 32 floating point registers, and 32 general purpose registers.

loop 1	hydro excerpt
loop 2	inner product
loop 3	inner product (no unrolling)
loop 4	banded linear equations
loop 5	tri-diagonal elimination, lower
loop 6	tri-diagonal elimination, upper
loop 7	equation of state excerpt
loop 8	p.d.e. integration
loop 9	integrate predictors
loop 10	difference predictors
loop 11	first sum
loop 12	first difference
loop 13	2-d particle pusher
loop 14	1-d particle pusher
linpack 1	index of the max. element
linpack 2	sum of the absolute values
linpack 3	scale and add
linpack 4	copy
linpack 5	inner product
linpack 6	Euclidean norm
linpack 7	apply plane rotation
linpack 8	construct plane rotation
linpack 9	scale
linpack 10	interchange

Table 1. The preliminary benchmark set.

2.3. The Benchmarks

We have the first 14 of the Livermore Loops and 10 of the linpack subroutines in our initial benchmark set. These benchmarks exhibit a wide range of data dependency patterns and resource usages. For example, loops 5 and 6 are sequential; whereas loops 2 and 7 are parallel. As another example, loop 4 has a low ratio of floating-point arithmetic operations to memory accesses; whereas loop 7 has a high ratio of floating-point arithmetic operations to memory accesses.

The benchmark performance of the *central* microarchitecture serves as the comparison basis for the other benchmark results presented in this paper. We use the executable codes generated for the *central* to estimate the performance. Table 2 shows the benchmark performance of the *central* microarchitecture with a 40ns cycle time. Even though the peak performance of the machine is

benchmark	central
lloop 1	19.98/44.05
lloop 2	8.93/32.14
lloop 3	7.14/28.57
lloop 4	5.00/22.50
lloop 5	4.41/20.59
lloop 6	4.41/20.72
lloop 7	12.04/43.52
lloop 8	13.64/45.08
lloop 9	12.14/47.14
lloop 10	5.23/46.51
lloop 11	3.57/28.57
lloop 12	7.90/48.69
lloop 13	4.44/36.69
lloop 14	5.60/17.91
linpack 1	2.94/20.59
linpack 2	16.00/41.00
linpack 3	8.70/48.91
linpack 4	NA/50.00
linpack 5	10.00/40.00
linpack 6	3.95/13.10
linpack 7	13.64/46.59
linpack 8	4.88/26.57
linpack 9	6.45/45.16
linpack 10	NA/50.00
average	8.23/36.03

Table 2. The estimated performance of the *central* microarchitecture.

25MFLOPS/50MIPS, the average benchmark performance is about 8.23MFLOPS/36.03MIPS†. Note that linpack routines 4 and 10 do not contain any floating point operations and they are excluded from the MFLOPS measure.

As for the MFLOPS measure, the difference between the peak and the average performance is due to two major factors. First, the extra processing (address calculation, memory access, loop control, etc.) which prevents the machine from executing one floating point arithmetic microoperation per cycle. For example, address calculations in Livermore loop 10 dominate the computation. Although the MFLOP count is low, the *central* microarchitecture executes Livermore loop 10 at almost peak performance. Second, the data dependencies prevent the compiler from issuing one floating point arithmetic microoperation per cycle. For example, Livermore loops 5 and 6 are sequential due to the data flow dependencies between array elements. In this case, the compiler simply can not make good use of the parallelism in the microarchitecture. As for the MIPS measure, the difference between the peak and the average performance is mainly due to data dependencies.

† The estimated performance numbers are optimistic due to the lack of cache models in that experiment. We have estimated the performance using a large cache with no valid initial contents (cold start); the average benchmark performance is about 4MFLOPS and 20MIPS.

2.4. Previous Works

The VLIW (very long instruction word) projects have conducted experiments using a compiler to exploit a highly parallel microarchitecture [11, 12, 13] crossing many chip and/or board boundaries. We focus on microarchitectures with one CPU chip and one FPU chip. There are three important implications from this difference: (1) we can reduce chip-crossing delays to achieve very short cycle time and small communication overhead between function units; (2) the cost of our microarchitecture is much lower than that of the VLIW microarchitectures; and (3) we do not have to unroll the loops many times before we can make good use of the microarchitectural parallelism. Due to practical technology constraints, the benefit of parallelism can be nullified by long clock cycle time, data transfer overhead, code explosion, and register spilling. Instead of putting the emphasis just on parallelism, we pay equal attention to the technology constraints.

Most of the microcode compaction works have suffered from the peculiarities of their target machines, e.g. microoperation selection and variable pipelines [14]. Our work is based on simple and regular microarchitectures so that we can focus on more important aspects such as register assignment and code scheduling, global analysis, and optimization. A clean implementation is possible due to the regular structure of the underlying microarchitecture.

The MIPS project at the Stanford University uses a compiler to exploit a pipelined microprocessor without hardware interlocks [5, 10]. The MIPS code reordering and register assignments are done in separate passes. In our research, code scheduling and register allocation are performed together so that just enough registers are used to exploit all the useful program parallelism. According to the results in Section 3.1, this is crucial for exploiting parallel microoperation issuing and execution unit pipelining.

The Architect's Workbench project at the Stanford University evaluates architectural design decisions for different applications [15]. The project, however, has not yet established microarchitecture models such as pipelining and parallel data paths to evaluate the performance directly. All the measurements available from the project are indirect parameters such as memory traffic ratio and cache hit/miss ratios. Instead of trying to evaluate the merit of various architectures, we use only one type of architecture so that we can focus on microarchitecture design decisions. Within this domain, we have established microarchitecture models to evaluate microarchitecture design decisions.

3. Experiments and Analysis

We have conducted experiments on our compiler's effectiveness at utilizing additional features in the microarchitecture. In particular, we estimate the changes to the benchmark performance when we modify the *central* microarchitecture. The main features we change are the microoperation issue bandwidth, the result distribution bandwidth, the pipeline structure of the execution units, and the number of data cache ports.

Due to the large number of possible combinations, we can not report experiments on all possible changes to the *central* microarchitecture. Most of the results reported here are derived by changing one of the features while keeping the others fixed. To illustrate the importance of balancing these features, we also report the joint effect of changing both the number of microoperations issued at each cycle and the pipeline structure of the memory access and floating point arithmetic units.

For each microarchitecture shown in the subsequent sections, the compiler generates optimized code sequences for that microarchitecture. The compiler takes a microarchitecture specification as input and generates code particularly scheduled for the specified microarchitecture. Therefore, the compiler generates optimized codes for a class of microarchitectures rather than just for one, as in the Bulldog Compiler [12]. Therefore, we can have a fair comparison among microarchitectures which differ in the degree of supported parallelism.

benchmark	no prelim.
lloop 1	0.59
lloop 2	0.83
lloop 3	0.99
lloop 4	0.76
lloop 5	0.95
lloop 6	0.90
lloop 7	0.55
lloop 8	0.45
lloop 9	0.40
lloop 10	0.72
lloop 11	0.88
lloop 12	0.67
lloop 13	0.68
lloop 14	0.85
linpack 1	1.00
linpack 2	0.51
linpack 3	0.67
linpack 4	0.56
linpack 5	0.57
linpack 6	1.00
linpack 7	0.46
linpack 8	0.97
linpack 9	0.69
linpack 10	0.61
average	0.72

Table 3. Relative benchmark performance without code reordering.

3.1. Preliminary Code Reordering

Before we report the effects of modifying the microarchitecture features, we would like to demonstrate the importance of performing preliminary code reordering. We have studied the effect of preliminary code reordering on the benchmark performance of the *central* microarchitecture. Table 3 shows the relative benchmark performance without preliminary code scheduling as compared to that with preliminary code reordering. Without preliminary code reordering, the benchmarks execute at 72% of the achieved speed. That is, by introducing preliminary code reordering, the *central* microarchitecture can potentially execute the benchmarks 39% faster.

Two major factors contribute to the performance difference. First, register recycling (visible for all the benchmarks) and spilling (visible for Livermore loops 8 and 13) introduce extra register and/or memory dependencies when we map the virtual registers to the physical registers. The extra dependencies can prevent the final code scheduler from effectively moving code around. Preliminary code

reordering reduces the desired distance of code movement during final code scheduling and therefore, reduces the impact of the extra dependencies. Second, preliminary code reordering reduces the lifetime of temporaries. After preliminary code reordering, the temporary values are produced as late as possible and consumed as early as possible. The probability of register spilling is reduced.

If a benchmark uses only a small number of variables and temporaries, one could employ a less aggressive register recycling policy so that fewer extra dependencies will be introduced. We decided not to implement this optimization for two reasons. First, without multiple register sets* [16], using fewer registers implies saving and restoring fewer registers before and after invoking the code. This can be a significant performance factor for large programs when global (and interprocedural) register allocation is implemented.

Second, most of the benchmarks do use many variables and temporaries after we unroll and fold the loops. For example, Livermore loop one uses 48 floating point variables and temporaries after we unroll and fold the inner loop, although the original version uses only 11. From the wide variety of the small benchmarks we have examined, **parallelized programs require significantly more temporaries than the sequential version¢**. Since we expect the problem to be even worse for large programs, we decided to implement a tight register allocation policy to avoid unnecessary register spilling.

Therefore, register recycling and spilling can introduce extra register and/or memory dependencies even for benchmarks which are small in their sequential form. Without preliminary code reordering, these extra dependencies decrease the chance of moving critical microoperations to their desired positions. Preliminary code reordering reorders the code according to the original data dependencies before introducing the extra dependencies. The critical microoperations have already moved close to the desired positions and they tend to stay in their neighborhood during final scheduling. Therefore, it is unlikely that the extra dependencies prohibit the desired movement of these critical microoperations.

With preliminary code reordering, the compiler can exploit almost all the useful parallelism with a reasonable number of floating point registers provided by the microarchitecture (32 in our case). From the experiments we have conducted so far, preliminary code reordering has performed effectively to achieve a good compromise between two conflicting goals: exploiting parallelism and minimizing register spilling.

3.2. Two Extreme Design Points.

The *basic* column in Table 4 shows the relative benchmark performance (normalized to that of the *central*) of the *basic* microarchitecture which differs from the *central* in two ways: (1) only one microoperation can be issues at each cycle, and (2) the floating point arithmetic units are not pipelined. The benchmark performance of the *basic* microarchitecture is about 59% of that of the *central*. A useful interpretation is that the benchmark performance increases by 70% from the *basic* microarchitecture to the *central*. Note that if we did not use preliminary code reordering (see Table 3), we would have gained only a small portion of the speedup.

benchmark	basic	maximal
lloop 1	0.35	1.23
lloop 2	0.59	2.24
lloop 3	0.69	1.01
lloop 4	0.79	1.04
lloop 5	0.96	1.00
lloop 6	0.96	1.00
lloop 7	0.40	1.41
lloop 8	0.35	1.65
lloop 9	0.27	1.25
lloop 10	0.61	1.02
lloop 11	0.76	1.01
lloop 12	0.51	1.46
lloop 13	0.51	1.13
lloop 14	0.84	1.00
linpack 1	0.84	1.07
linpack 2	0.37	1.25
linpack 3	0.50	1.86
linpack 4	0.50	1.20
linpack 5	0.48	1.94
linpack 6	1.00	1.05
linpack 7	0.31	1.97
linpack 8	0.59	1.10
linpack 9	0.50	1.54
linpack 10	0.50	1.35
average	0.59	1.32

Table 4. The relative performance of two extreme designs.

The *maximal* column in Table 4 shows the relative performance of a microarchitecture without resource limitations. Such a microarchitecture performs strictly inversely proportional to the depth of the data dependency graph. The *maximal* microarchitecture performs about 32% better than the *central* †.

3.3. The Microoperation Issue Bandwidth

All the microarchitectures in this study have one pipelined data cache access port and one pipelined floating-point arithmetic unit. The only variation is in the number of integer ALU microoperations which can be issued per cycle. We do not vary the number of floating point arithmetic units and the number of memory ports; we explore only one dimension of the design space.

The *one* column in Table 5 shows the relative performance of a microarchitecture in which one microoperation is issued per cycle and there is only one integer ALU. The *one* microarchitecture can be derived from the *basic* microarchitecture by linearly pipelining the cache access and the floating point arithmetic units. Notice that using the *basic* microarchitecture as the comparison basis, pipelining increases the performance by 17%.

The advantage of pipelining in the *one* microarchitecture is somewhat disappointing. Only one microoperation can enter the machine every cycle. This narrow path forces all the execution units to

*We have decided not to implement multiple register sets because of their poor utilization of precious register resource. Implementing multiple register sets to support nested procedure calls of reasonable depth usually implies that each procedure sees only a small fraction of the registers implemented in the underlying hardware.

¢The problem of creating large number of temporaries in parallelized programs has been reported in [11].

† We would like to emphasize that the performance ratios must be qualified by the current status of our code optimization techniques.

benchmark	one	three
lloop 1	0.57	1.05
lloop 2	0.71	1.32
lloop 3	0.85	1.00
lloop 4	0.84	1.00
lloop 5	0.96	1.00
lloop 6	0.96	1.00
lloop 7	0.57	1.17
lloop 8	0.57	1.14
lloop 9	0.52	1.17
lloop 10	0.61	1.00
lloop 11	0.76	1.00
lloop 12	0.51	1.34
lloop 13	0.64	1.00
lloop 14	0.98	1.00
linpack 1	0.84	1.06
linpack 2	0.61	1.09
linpack 3	0.52	1.41
linpack 4	0.50	1.20
linpack 5	0.62	1.20
linpack 6	1.00	1.05
linpack 7	0.54	1.28
linpack 8	0.73	1.01
linpack 9	0.56	1.19
linpack 10	0.50	1.15
average	0.69	1.12

Table 5. The relative performance for various microoperation issue bandwidth

be idle most of the time. The microinstruction format becomes the major bottleneck limiting the concurrency exploited by the microarchitecture.

When we allow two microoperations to be issued in each microinstruction as well as pipeline the execution units (the *central* microarchitecture), the performance increased by 70% (from 59% to 100%). For the benchmarks shown in this paper, pipelining the execution units without increasing the microoperation issuing bandwidth is not a cost-effective approach to improve performance.

The *three* column in Table 5 shows the relative performance of a microarchitecture allowing three microoperations in each microinstruction. The *three* microarchitecture can be derived from the *central* by adding one integer ALU field to each microinstruction. We keep the number of floating point arithmetic or memory access microoperations unchanged so that we only make changes in one of the four dimensions at a time. For example, the machine can issue one floating point arithmetic, one memory access, and one integer ALU microoperation every cycle.

The benchmark performance of the *three* microarchitecture is about 12% higher than the *central* microarchitecture. By examining the schedule for individual benchmarks, we have found two major factors which limit the performance improvement. First, memory access and floating point arithmetic units become the bottleneck. Second, the optimization techniques implemented in the code generator have difficulty in finding parallel microoperations to be scheduled into the extra ALU field.

To support a large number of microoperations in each microinstruction, enough register access ports must be available for the microoperations to fetch their input operands and to store their execution results. A large number of register access ports will either require more chip area and access time (due to increased word-lines/bit-lines or multiple copies) or increase the number of clock phases (due to sequencing the accesses). Therefore, the increased number of microoperations in each microinstruction must be justified by a significant decrease in the schedule's length.

Using the present code generator, it is difficult to justify an increase from two to three microoperations in each microinstruction. Even with better code generators, we need to increase the number of memory access ports and floating point arithmetic units to make good use of the increased microoperation issue bandwidth.

benchmark	single	unlimited
lloop 1	0.65	1.00
lloop 2	0.78	1.00
lloop 3	0.97	1.00
lloop 4	0.96	1.00
lloop 5	1.00	1.00
lloop 6	1.00	1.00
lloop 7	0.61	1.00
lloop 8	0.59	1.01
lloop 9	0.54	1.03
lloop 10	0.71	1.00
lloop 11	0.86	1.00
lloop 12	0.60	1.00
lloop 13	0.67	1.07
lloop 14	0.97	1.00
linpack 1	1.00	1.00
linpack 2	0.55	1.00
linpack 3	0.57	1.00
linpack 4	0.61	1.00
linpack 5	0.58	1.00
linpack 6	1.00	1.00
linpack 7	0.61	1.07
linpack 8	0.87	1.00
linpack 9	0.65	1.00
linpack 10	0.61	1.00
average	0.75	1.01

Table 6. The relative performance for various result distribution bandwidth.

3.4. The Result Distribution Bandwidth

The *single* column in Table 6 shows the relative performance of a microarchitecture obtained by reducing the result distribution bandwidth of the *central* to one result per cycle. This is similar to the Common Data Bus in the floating point unit of the IBM 360/91 [17]. By adding one bus to both the CPU and the FPU chips, we improve the benchmark performance by 33% (from 75% to 100%).

The *unlimited* column in Table 6 shows the relative performance of a microarchitecture obtained by giving the *central* unlimited result distribution bandwidth. The benchmark performance showed almost no improvement. Note, however, that if we added more memory ports, floating point arithmetic units, and higher microopera-

tion issue bandwidth, the increased result distribution bandwidth would increase performance more significantly. For the execution resources in the *central* microarchitecture, providing two result distribution buses each chip seems to be a balanced design.

benchmark	(3,3)	neither	n-cache	n-float
lloop 1	1.11	0.35	0.91	0.35
lloop 2	1.16	0.58	0.87	0.59
lloop 3	1.68	0.69	0.96	0.71
lloop 4	1.37	0.93	1.00	0.93
lloop 5	1.53	0.99	1.00	0.99
lloop 6	1.51	0.99	1.00	0.99
lloop 7	1.12	0.45	0.86	0.46
lloop 8	1.04	0.38	0.82	0.38
lloop 9	1.03	0.39	0.95	0.41
lloop 10	1.02	0.95	0.95	1.00
lloop 11	1.16	0.97	0.97	1.00
lloop 12	1.00	0.72	0.89	0.74
lloop 13	1.15	0.89	0.92	0.90
lloop 14	1.24	0.89	0.96	0.92
linpack 1	1.13	1.00	1.00	1.00
linpack 2	1.04	0.41	0.89	0.42
linpack 3	1.00	0.58	0.87	0.57
linpack 4	1.00	0.79	0.79	1.00
linpack 5	1.11	0.52	0.81	0.52
linpack 6	1.26	1.00	1.00	1.00
linpack 7	1.07	0.33	0.87	0.33
linpack 8	1.12	0.61	0.74	0.79
linpack 9	1.11	0.59	0.99	0.59
linpack 10	1.00	0.82	1.00	0.82
average	1.17	0.70	0.92	0.73

Table 7. The relative performance for various function unit pipeline structures.

3.5. The Function Unit Pipeline Structure

The *(3,3)* column in Table 7 shows the relative benchmark performance of a microarchitecture in which the floating point addition and the floating point multiplication both take 3 cycles. Otherwise, the *(3,3)* microarchitecture is the same as the *central*. The benchmarks which benefit the most from the reduced pipeline latency are the more sequential ones. Because the parallel benchmarks have enough independent microoperations to hide the pipeline latency, they are much less sensitive to the change of that latency.

A floating point adder with a pipeline latency of 4 cycles and a multiplier with a pipeline latency of 6 cycles require reasonable area on the FPU chip. With a potential 17% performance improvement, we are investigating the feasibility of implementing floating point adders and multipliers with a latency of 3 cycles. In addition, with identical latency, the adder and the multiplier can share the same result bus without conflict resolution logic. This simplifies the design of one of the result bus on the FPU chip. The design of the other FPU result bus, dedicated to the memory access port, is simple to begin with.

The *neither* column in Table 7 shows the relative performance of a microarchitecture derived from the central microarchitecture by removing pipelining from both the floating-point arithmetic unit and the cache access port. All the speedup achieved by the *neither* microarchitecture over the *basic* microarchitecture is due to the increased microoperation issue rate. A possible way to interpret the result is that pipelining contributes to 30% of the benchmark performance in the *central* microarchitecture.

The *n-cache* column in Table 7 shows the relative performance of a microarchitecture derived from the *central* by removing pipelining from the cache access port. The *n-float* column in Table 7 shows the relative performance of a microarchitecture derived from the *central* by removing pipelining from the floating-point arithmetic unit. Note that for the benchmarks presented here, pipelining the floating point arithmetic unit is significantly more important than pipelining the data cache access.

3.6. The Number of Data Cache Access Ports

We have investigated the impact of adding one more data cache port to the *central* microarchitecture. The average benchmark performance increased by only 1%. By examining the schedules in more detail, we found that data memory bandwidth is not the limiting factor of performance of the *central* microarchitecture. A dedicated memory port provides a good balance between the execution bandwidth and the data memory bandwidth. We will have to increase the program parallelism visible to the compiler and increase the microoperation issuing bandwidth before we can observe significant benefit due to the additional data cache port.

benchmark	(1,F)	(1,T)	(2,F)	(3,F)	(3,T)
lloop 1	0.35	0.58	0.35	0.35	1.10
lloop 2	0.59	0.71	0.58	0.59	1.32
lloop 3	0.69	0.85	0.69	0.69	1.00
lloop 4	0.79	0.84	0.93	0.93	1.00
lloop 5	0.96	0.96	0.99	0.99	1.00
lloop 6	0.96	0.96	0.99	0.99	1.00
lloop 7	0.40	0.57	0.45	0.46	1.17
lloop 8	0.35	0.57	0.38	0.38	1.13
lloop 9	0.27	0.52	0.39	0.42	1.17
lloop 10	0.61	0.61	0.95	0.95	1.00
lloop 11	0.76	0.76	0.97	0.97	1.00
lloop 12	0.51	0.51	0.72	0.75	1.34
lloop 13	0.51	0.64	0.89	0.88	1.00
lloop 14	0.84	0.98	0.89	0.89	1.00
linpack 1	0.84	0.85	1.00	1.06	1.06
linpack 2	0.37	0.61	0.41	0.42	1.09
linpack 3	0.50	0.52	0.58	0.59	1.41
linpack 4	0.50	0.50	0.79	0.80	1.20
linpack 5	0.48	0.62	0.52	0.52	1.21
linpack 6	1.00	1.00	1.00	1.05	1.05
linpack 7	0.31	0.54	0.33	0.33	1.28
linpack 8	0.59	0.73	0.61	0.60	1.01
linpack 9	0.50	0.56	0.59	0.60	1.19
linpack 10	0.50	0.50	0.82	0.79	1.15
average	0.59	0.69	0.70	0.71	1.12

Table 8. The relative performance of combinations of pipelining and microoperation issue bandwidth.

3.7. Pipelining and Microoperation Issue Bandwidth

Many interesting experiments can be conducted to identify the balanced design points for different performance goals. There are, however, exponentially many experiments to cover all possible design points. Due to limited space, we show only one of these experiments which illustrate the joint effect of varying more than one design parameters at the same time. This experiment illustrates how the execution unit pipelining and microoperation issuing bandwidth should be balanced to increase the benchmark performance.

Each column in Table 8 is marked by a number (1, 2, or 3) and a flag (T or F). The number specifies the number of microoperations issued per cycle. The flag specifies whether the arithmetic units are pipelined. The other design features are the same as the *central*. We can identify the following correspondence:

(1) Microarchitecture *(1,F)* is the *basic*;

(2) Microarchitecture *(1,T)* is the *one*;

(3) Microarchitecture *(2,T)* is the *central*†;

(4) Microarchitecture *(3,T)* is the *three*.

The *(2,F)* and *(3,F)* microarchitectures have not been introduced in the previous sections.

The numbers shown in Table 8 strongly suggest that execution unit pipelining and microoperation issue bandwidth must be improved together to effectively improve benchmark performance. Improving the the microoperation issue bandwidth from one to three microoperations per cycle alone enhanced the benchmark performance by only 20%. Pipelining the execution units alone enhances the benchmark performance by only 17%. When we improve both, from *basic (1,F)* to *central (2,T)*, we improve the benchmark performance by 70%.

4. Concluding Remarks

By generating optimized microcode for a class of parallel microarchitectures, we have been able to evaluate the effect of each additional degree of parallelism in the microarchitecture. The class of microarchitectures are so simple and regular that we can concentrate on the parallelism aspect. Experiments have been performed on the impact of microinstruction issue bandwidth, the execution unit pipeline structure, the data cache access bandwidth, and the execution result distribution bandwidth. The joint effect of the microinstruction issue bandwidth and the execution unit pipeline structure has been used to illustrate how these design parameters should be balanced to achieve high performance.

We are designing and implementing a parallel microarchitecture consisting of a CPU chip, a FPU chip, a chip containing the data store of the cache, and a bus interface chip. It is closely modeled by the *central* microarchitecture, although the floating point arithmetic pipeline latency is still an issue. At this early design stage, the versatile compiler has been a very effective tool for evaluating design decisions.

Ultimately, we would like to automatically generate a well balanced microarchitecture given a set of workloads written in C and a set of technological constraints. Variations of this approach has received increasing attention. For example, the Architect's Workbench project at the Stanford University has been integrating a versatile optimizing compiler with instruction set level simulation tools to evaluate design decisions at the instruction set level.

The performance numbers reported from the Bulldog compiler project [12] suggest that unrolling the loop eight to sixteen times (rather than two times in our case) followed by applying the trace scheduling technique can potentially extract a much higher degree of parallelism than reported in this paper. We are conducting experiments to evaluate the Bulldog approach considering the effects of code explosion (reduced instruction buffer/cache performance), register usage explosion (register spilling and/or data movement between register banks), and cycle time penalty (multiboard implementation due to technology constrains).

We are collecting, compiling, and analyzing a large number of benchmarks such as the UNIX† system code, CAD tools, the UNIX mathematical libraries, the statistical packages, and the signal processing packages. Numerous experiments must be performed before strong statements can be made regarding the code generator's ability to exploit parallel microarchitectures. We are also conducting more experiments on the joint effect of more design parameters.

With the advancing fabrication technology, microarchitectures supporting a high degree of parallelism have become a cost effective way to achieve high performance. More experiments will have to be performed before we can address all the issues regarding the integration of the compiler with the microarchitecture to achieve well balanced high performance designs. The results we have reported in this paper show that such an integration can be effectively done and can eventually contribute to the automatic generation of application specific high speed processors.

Acknowledgements

The authors would like to acknowledge Nancy Warter, Sharon Simonson, Sadun Anik, and the other members of the Computer System Group for their invaluable comments and suggestions.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[2] F.H. McMahon, *L.L.N.L. FORTRAN KERNELS: MFLOPS*, Lawrence Livermore National Laboratories, 1984.

[3] J.J. Dongarra and et.al. *LINPACK Users' Guide*, the Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[4] D.J. Kuck, *et al.*, "Dependency Graphs and Compiler Optimizations", *Proceedings of 8th POPL*, Jan. 1981.

[5] J.L. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Trans. on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 422-448.

[6] W.W. Hwu and P.P. Chang, "Code Generation Techniques for Exploiting Single Chip Parallel Microarchitectures," Computer Systems Group Report, University of Illinois, in preparation.

[7] W.W. Hwu and Y.N. Patt, "HPSm2: a Refined Single-chip Microengine," *Proceedings of the* 21st Annual Hawaii Interna-

† We omitted the (2,T) column since the contents are all 1.0.

† UNIX is a trademark of Bell Laboratories.

tional Conference on System Sciences, 1988.

[8] W.W. Hwu and Y.N. Patt, "Exploiting Horizontal and Vertical Concurrency via the HPSm Microprocessor," *Proceedings of the 20th International Microprogramming Workshop*, Colorado, December, 1987.

[9] W.W. Hwu, *HPSm: Exploiting Concurrency to Achieve High Performance in a Single-chip Microarchitecture*, Ph.D. Dissertation, Computer Science Division, University of California, Berkeley, December, 1987.

[10] J.L. Hennessy, *et al.*, "The MIPS Machine," COMPCON, IEEE, Spring 1982.

[11] A. Nicolau and J.A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," IEEE Transactions on Computers, vol. c-33, no232

[12] Ellis, J.R., *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1986.

[13] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, vol. c-30, no. 7, July 1981, pp. 478-490.

[14] M. Tokoro, E. Tamura, and T. Takizuka, "Optimization of Microprograms," IEEE Transactions on Computers, vol. c-30, no. 7, July 1981, pp. 491-504.

[15] C.L. Mitchell and M. Flynn, "A Workbench for Computer Architects", *IEEE Design & Test of Computers*, February 1988.

[16] R.J. Eickemeyer and J.H. Patel, "Performance Evaluation of Multiple Register Sets," *The 14th Annual International Symposium on Computer Architecture Conference Proceedings*, Pittsburgh, Pennsylvania, June 2-5, 1987.

[17] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, January 1967, pp 25-33.