Systematic Register Bypass Customization for Application-Specific Processors

Kevin Fan, Nathan Clark, Michael Chu, K. V. Manjunath, Rajiv Ravindran, Mikhail Smelyanskiy, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{fank, ntclark, mchu, kvman, rravindr, msmelyan, mahlke}@umich.edu

Abstract

Register bypass provides additional datapaths to eliminate data hazards in processor pipelines. The difficulty with register bypass is that the cost of the bypass network is substantial and grows substantially as processor width or pipeline depth are increased. For a single application, many of the bypass paths have extremely low utilization. Thus, there is an important opportunity in the design of application-specific processors to remove a large fraction of the bypass cost while maintaining performance comparable to a processor with full bypass. To this end, we propose a systematic design customization process along with a bypass-cognizant compiler scheduler. For the former, we employ iterative design space exploration wherein successive processor designs are selected based on bypass utilization statistics combined with the availability of redundant bypass paths. Compiler scheduling for sparse bypass processors is accomplished by prioritizing function unit choices for each operation prior to scheduling using global information. Results show that for a 5-issue customized VLIW processor, 70% of the bypass cost is eliminated while sacrificing only 10% performance.

1 Introduction

The next generation of embedded processors will need to perform computationally demanding processing of images, sound, and video while achieving both low cost and minimal power dissipation. Application-specific hardware design provides an effective strategy to meet these challenging demands. With application specific hardware, the design is customized to specifically suit the computation needs of the application.

Hardware accelerators in the form of ASICs are the most common form of application-specific hardware used today. ASICs are generally nonprogrammable, hardwired solutions to a point problem, such as MPEG decompression. The central problem with ASICs is the lack of flexibility in terms of post-programmability (the ability to modify the application after the hardware is constructed) and reusability (the ability to utilize the hardware for multiple similar applications). Application-specific processors can potentially bridge the gap between the flexibility offered by a general-purpose processors and the speed and power savings offered by an ASIC.

The key to designing effective application-specific processors lies in finding productive opportunities for customization. There are a myriad of architectural decisions to be made when designing a processor, including memory configuration; fetch and issue logic; the number, type, and connectivity of function units (FUs) and register files; and pipeline depth. Each of these can potentially be customized. Ad-hoc hardware customization is inefficient in terms of design cost and time to market. Thus, systematic strategies must be developed to accomplish customization.

In this paper, we focus on one particular portion of application-specific processor design: the register bypass logic. *Register bypassing*, or data forwarding, is a technique for eliminating data hazards in pipelined processors. With bypassing, additional datapaths and control logic are added so that an operation's result is available for subsequent operations before it has been written to an architectural register, thereby reducing the effective latency of operations. Bypassing was first introduced in the IBM Stretch and is considered a standard part of all modern pipelined processors [5].

A fully bypassed processor allows any FU to read its inputs from any of the subsequent pipeline stages. The cost of implementing a full bypass network is significant, both in terms of area and wire delay [17]. For instance, the Alpha 21064 has 45 separate bypass paths [16]. As the number of FUs increases or as pipeline depth grows, bypass cost increases substantially. However, many of the bypass paths are underutilized or not used at all for any specific application. Therefore, there is opportunity to construct a processor with partial bypass that will have substantial cost savings and comparable performance to a processor with full bypass.

There are two central challenges in building processors with customized bypass networks: systematic identification of the Pareto-optimal design points and effective compilation. Systematic design of partial bypass processors is difficult because the design space is very large. Not only are there a large number of possible processor configurations, but it is not obvious whether a certain design will perform better or worse than another for a given application. For example, removing a highly utilized bypass could have little effect on overall performance if other bypass paths between compatible FUs are available, or if the additional latency can be overlapped with other computation. However, removing a less-utilized bypass may result in significant performance degradation if the additional latency cannot be tolerated.

Our approach is to iteratively explore the design space beginning with a fully bypassed processor. Dynamic utilization of bypass paths and the availability of redundant bypass paths are used to select candidates for removal. Iterative pruning of bypasses and re-compilation are used to identify the processors with the best achievable tradeoff in terms of cost and performance.

The second challenge is effectively compiling applications to a processor with partial bypass. The focus is on VLIW processors wherein it is the compiler's job to perform instruction scheduling. With full bypass, all operation latencies are constant values, unlike partial bypass, where the effective operation latencies vary based on FU assignment. That is, the latency of each producer-consumer edge depends on the particular FUs assigned to both the producer and consumer, and what bypass paths exist between those FUs. This causes a problem with compiler schedulers because they make locally greedy decisions for assigning operations to FUs. Each operation is scheduled on an FU that minimizes the operation's schedule time. However, this choice may preclude subsequent operations in a chain of dependent operations from having a bypass path available to use. As a result, performance may suffer due to longer schedules.

To manage this problem, we employ a more intelligent strategy of assigning operations to FUs. Prior to scheduling, paths of dependent operations are examined, in their entirety, for FU selection. It can be counter-productive to eliminate all FU assignment flexibility in the scheduler, though, so our algorithm merely prioritizes compatible FUs for each operation. Scheduling is performed cognizant of the prioritization, but the final FU assignment decision is left to the scheduler.

2 Background and Motivation

The architectural model targeted in this paper is a pipelined VLIW processor, shown in Figure 1. Bypass logic connects the inputs of the FUs with the results of subsequent pipe stages. The pipeline latch from each pipeline stage after execute then forwards values back to operand multiplexors (MUXes) which are placed prior to each FU. The cost of the bypass logic primarily stems from two sources: the datapath logic and the control logic. The datapath logic consists of result buses which are used to broadcast values to the execution stage. Operand MUXes, with a fan-in of the total number of result buses, gate the correct operands to the FUs. In addition to the datapath requirements, control logic is sent back with each result in order to select the appropriate input to the MUXes. Another source of complexity is the use of predication. Predication nullifies certain instructions based on the values of previously calculated predicates. The values of predicates super-

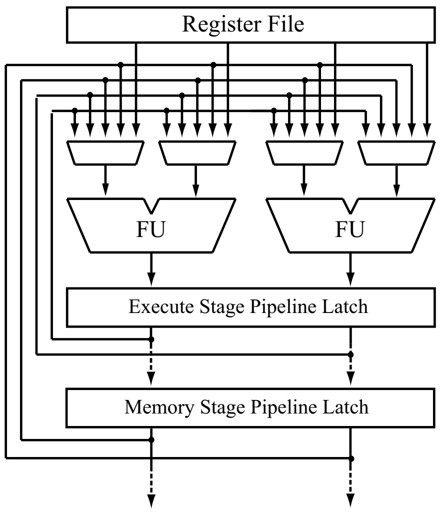


Figure 1. Our processor model.

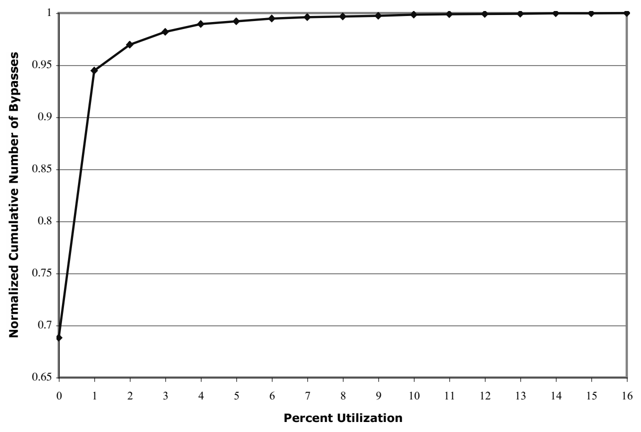


Figure 2. Cumulative distribution of used bypasses.

sede the control logic of whether or not bypasses are taken. In addition, predicates can themselves be bypassed.

Wider issue increases the number of instructions requiring bypass per stage, and deeper pipelines increase the number of stages requiring bypass. For example, given an issue width i and n stages after execution to bypass from, the full-bypass model on two-input FUs would require $(2 \times i^2 \times n)$ bypass paths. Thus, a 4-issue 5-stage pipeline would require $(2 \times 4^2 \times 3) = 96$ bypass paths.

Palacharla's work [17] showed that the delay of the bypass logic is largely dependent on the time it takes to drive the output of an FU to the corresponding result wire, and thus is dependent on the length of the wire itself. Since the length of the result wires is a function of the issue width, wider issue machines can significantly increase bypass delay. Palacharla shows that this delay grows quadratically with issue width.

Our study focuses on reducing the complexity involved in having a large bypass network for wide-issue, deep-pipelined machines by eliminating bypass paths that are not used or seldom used. Figure 2 shows the cumulative distribution of the average utilization of individual bypasses across the MediaBench benchmarks [14]. The utilization data represents the average number of cycles the bypass is active over the entire run of each benchmark. On the processor model studied, a 10-issue, 5-stage pipeline, there are a total of $(2 \times 10^2 \times 3) = 600$ bypass paths required for a fully bypassable network. The figure shows that approximately 69% of all bypasses were never used at all. In addition, 94% of bypasses are utilized only 1% of the time. This data clearly shows that there is an important opportunity to trade a small performance loss for large reductions in cost and complexity of the bypass network for an application specific processor. Many of the minimally used bypasses can be targeted for removal.

Related Work. Application-specific processors have been extensively studied in the past. Two examples are CustomVLIW [10] and Cryptomaniac [21]. These works showed that application-specific designs can achieve large design wins in image processing and cryptography. Automated design through design space exploration has also been investigated in several research projects, including MIMOLA [15], TTA [12], PICO [3], and NOVA [13]. A retargetable compiler combined with an architecture exploration system facilitates automatic design of application-specific processors.

Ahuja et al. [4] performed the initial study on the impact of partial bypass. Their study focused on detailing the amount of performance lost due to interlock stalls from missing bypasses on a scalar processor. Recently, Brown and Patt [6] studied the effect of limited bypass on pipelined adders and register files. They concluded that for multi-level bypass networks, large fan-in input MUXes increase cycle time. Removing one level of bypass paths results in 1-3% degradation in performance.

Many studies have been performed on the bypass networks of VLIW processors. Abnous and

Bagherzadeh [1] studied the effects of bypassing among FUs in their VIPER VLIW processor. They concluded that a fully connected bypass network does not provide performance improvement when considering cycle time penalty and required silicon area. Cohn et al. [8] did a study of a partial bypass configuration of the iWarp VLIW processor, concluding that partial bypass helps reduce cost with negligible reduction in performance.

The work by Buss et al. [7] is closely related to our study. They aim at reducing inter-cluster copy operations by placing operand chains into the same cluster and assigning FUs to clusters such that the inter-cluster communication pattern is minimized. They insert bypass paths between the most frequently communicating datapaths. Unlike theirs, we begin with a full bypass network and use a heuristic driven pruning of bypass paths. Also, we expose the bypass paths to the compiler and use intelligent scheduling techniques to effectively use a partially bypassed machine.

3 Bypass Customization

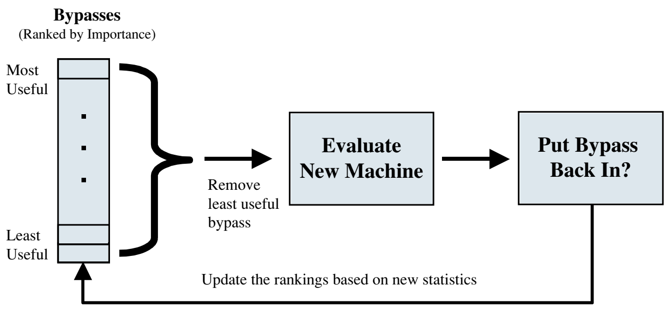
When dealing with any large design space, clearly it is impossible to exhaustively explore all potential designs. For example, if we were to explore just the partial bypass associated with the processor examined in Section 5, we would have to explore 2^{600} possible designs. To deal with this complexity, a systematic design strategy must be employed.

Our bypass customization system consists of three central components: design space explorer, processor synthesizer, and retargetable compiler. The head of the system is the design space explorer, henceforth referred to as the *explorer*. The explorer provides a machine specification describing which bypass paths are available. This specification can be used by the *synthesizer* to generate HDL for a processor and create a high level machine description, or *mdes* [2], which is used to retarget the compiler. Our system uses *elcor*, part of the Trimaran Research Infrastructure [20], as the retargetable compiler. Elcor is used to generate statistics, such as the utilization of each bypass path and the cost of the machine, to guide the explorer in picking which bypasses to remove. The explorer, synthesizer, and compiler form a feedback loop reminiscent of the PICO [19], TriMedia [11], and Delft [12] design flows, among others. It is important to note that none of these strategies use resource utilization to guide the direction of their space exploration. Approaches like those proposed in [18] only use resource utilization to create a design, and do no exploration.

The strategy our design space explorer uses would best be described as *profile guided Pareto ascent*, or PGPA, and is displayed graphically in Figure 3. Initially, we start with a full bypass machine. We then compile to that machine and get statistics on the utilization of each bypass path (a bypass path is uniquely defined by a producing resource, an output port from the producer, a consuming resource, an input port on the consuming resource, and a pipeline latch that the data is read from) and the number of redundant bypass paths that were available at the same scheduled cycle. Using this information, we choose a single bypass path to remove, and then regenerate the cost, performance, and utilization statistics. If the machine lost too much performance from the removal of the bypass path, then that path is placed back in the machine and another path is chosen. This strategy is called "Pareto ascent" because we start out with the machine that takes the fewest number of cycles to execute and hence has the highest cost. Then, the explorer moves up the Pareto to machines that take more cycles to execute but have lower cost.

This strategy differs from Pareto descent as described in [19] for two reasons. First, we are not branching out in all directions. Pareto descent does not intelligently add neighbors; it simply incrementally modifies each design variable (e.g., adds a single FU or a quanta of registers). PGPA uses utilization statistics garnered from a profile to intelligently pick a single direction. This greatly improves runtime at the possible cost of missing Pareto points from exploring a smaller part of the design space. The second difference between these two strategies is that PGPA subtracts resources as opposed to adding them. This makes sense for bypass customization because we cannot easily gather statistics on which non-existent bypass paths could have been used.

As mentioned previously, the compiler returns two pieces of information for each bypass path: the number of dynamic utilizations and the number of redundant bypasses available. To clarify,



$$\textbf{Importance} = \frac{(\textit{dynamic utilizations})^2}{(\textit{total dynamic cycles}) \times (\textit{number of redundant bypasses} + \varepsilon)}$$

Figure 3. The design space explorer process.

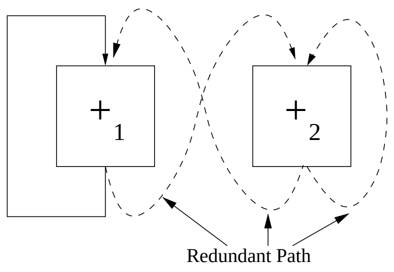


Figure 4. Redundant bypass paths.

given a bypass path P from FU A to FU B, a redundant bypass path Q is one that runs from FU A′ to FU B′, where A′ and B′ are functionally equivalent to A and B, respectively, and the bypass path Q is not utilized during the same cycle P is used. For example, consider the two adders $+_1$ and $+_2$ shown in Figure 4. The bypass path from $+_1$ to $+_1$ (solid line) has three potential redundant paths: $+_1$ to $+_2$, $+_2$ to $+_1$, and $+_2$ to $+_2$ (dotted lines). On a given cycle, there is a redundant path with $+_1$ to $+_1$ if any of the potential redundant paths exist and are unused. For our statistics, at most one redundant bypass is counted for each dynamic use of a bypass path, so that the maximum number of redundant bypasses is equal to the number of dynamic utilizations.

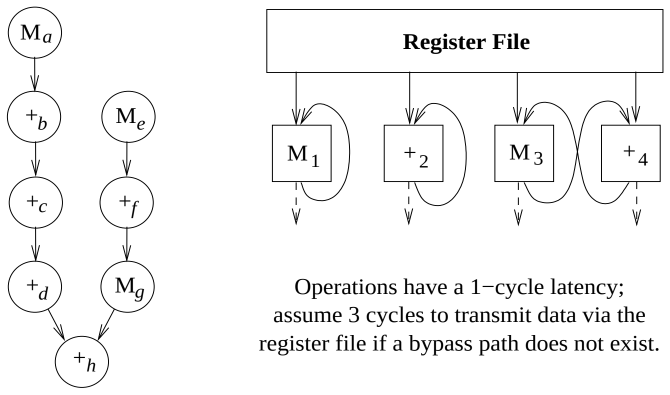
Note that the existence of a redundant bypass is a better metric for determining whether a bypass is removable than whether an equivalent FU is free at the time of scheduled use. Even if an equivalent FU is not free, if we were to remove the bypass in question, it is likely that the operations can swap their FU assignments during that cycle and make use of the redundant bypass.

Given the dynamic utilizations and the number of redundant bypasses for each bypass path, we can define the *percentage utilization* of each bypass as dynamic utilizations/total dynamic cycles in the program, and the *offload potential* of a bypass as the number of redundant bypasses/dynamic utilizations. The importance heuristic that we use to rank bypasses is the quotient of these two numbers, *percentage utilization* /*offload potential*, resulting in the overall equation for importance as shown in Figure 3. We add ϵ to the redundant bypasses to account for the fact that scheduling flexibility slightly increases the effective offload potential. Intuitively this heuristic makes sense: if a bypass is never used, it has an importance of zero; if all bypasses have a proportional number of redundant bypasses available, the least utilized bypass will have the lowest importance. We empirically verified that the heuristic makes intelligent selections by comparing it against selections made by an exhaustive algorithm, but these results are omitted due to space constraints.

One defining characteristic of our design exploration strategy is that it can only remove one bypass path per iteration. This is intentional, primarily because of the complex interplay between separate bypass paths and the schedule. Take for example if we removed two bypass paths from equivalent resources. This would effectively invalidate our redundant utilization statistics, since each bypass contributed to the redundant bypass count of the other. Removing both would have an unpredictable and decidedly negative impact on the schedule length. Another scenario involves removing one bypass path which causes the utilizations of a second bypass path to be moved to a different cycle in the schedule. Again, this invalidates our redundant utilization statistics, and leads to an unpredictable schedule.

4 Compilation for Partial Bypass

This section discusses our strategy for compiling to a machine with partial bypass. First, we show that blindly applying list scheduling can yield poor results. Then, we describe our modifications to



Operations have a 1−cycle latency; assume 3 cycles to transmit data via the register file if a bypass path does not exist.

Figure 5. Example DFG and partially-bypassed machine.

efficiently compile for a machine with partial bypass.

4.1 Problems with Conventional Scheduling

Consider the machine and dataflow graph (DFG) shown in Figure 5. The latencies on the edges of the DFG determine the critical path, which in turn determines the priority of operations. In the case of a machine with full bypass, the latency on an edge depends only on the latency of the FU that executes the producer. However, in a machine with partial bypass, this latency could be amplified due to absence of a bypass path. Consequently, the latency of an edge varies based on the assignment of operations to FUs. In our system, we initially take an optimistic approach to determine the latency of an edge. We set the latency of an edge to be the minimum over all possible bindings of the producer and consumer operations to FUs. These edge latencies are used for calculating operation priorities.

The scheduler looks at operations in priority order, trying to schedule each at the earliest possible time. It picks the earliest time in which the operation can be legally scheduled and an FU is available. When more than one FU is free, it picks one arbitrarily. Note that this decision will not affect the final schedule when scheduling for a machine with full bypass. Table 1(a) shows the schedule resulting from arbitrary choices of FUs for operations in the machine with partial bypass. The achieved schedule is very sparse due to the lack of intelligent placement. Two important mistakes are made. First, operations a and e should have their FU assignments transposed. Second, f should be placed on $+_4$ rather than $+_2$. However, these choices require looking ahead in the schedule to determine the effects on future operations. Naïve list schedulers do not perform lookahead, thus poor decisions are made.

4.2 Compiling for a Machine with Partial Bypass

To compile for a machine with partial bypass, we insert a new *FU prioritization* phase which takes place prior to scheduling. This phase annotates each operation with an ordered list of preferred FU alternatives. These alternatives are used as hints for the scheduler when choosing FUs. When none of the FUs in the list are free at the time the scheduler is trying to schedule an operation, it falls back to arbitrarily picking a free FU. The next two sections describe two different approaches for generating these hints.

Bottom-Up Greedy FU Assignment. One method we use to prioritize FUs is similar to the Bottom-Up Greedy (BUG) clustering algorithm [9]. BUG is a well-known cluster assignment algorithm which recurses up through the DFG in a depth-first fashion, from exit to entry operations, critical paths first, passing along estimates of good cluster assignments for each operation. As the algorithm returns back down the DFG, it makes final assignments of operations to clusters such that each operation completes soonest. Our FU assignment algorithm is analogous to this process,

time	M_1	$+_2$	M_3	$+_4$
0	e		a	
1				b
2				
3				
4		f		
5		c		
6		d		
7				
8	g			
9				
10				
11				
12		h		

(a)

time	M_1	$+_2$	M_3	$+_4$
0	a		e	
1				f
2			g	
3				
4		b		
5		c		
6		d		
7		h		

(b)

time	M_1	$+_2$	M_3	$+_4$
0	e		a	
1				b
2				
3				
4				f
5		c	g	
6		d		
7				
8				
9		h		

(c)

Table 1. Schedule generated by (a) bypass-unaware scheduler, (b) BUG FU prioritization and (c) ILP algorithm for the example DFG.

assigning operations to FUs depending on bypasses available in the machine. Note that in this context, "assignment" refers to the generation of hints for the scheduler; ultimately, the scheduler may choose a different FU.

In order to decide which FUs are good candidates for a certain operation, the algorithm calculates a *completion time*—the time when the operation would complete if executed by a given FU—for each potentially valid FU. This *completion time* takes into account the time to get input operands via bypass paths or register files; time to execute the operation; and time to transmit results to potential consumers via bypass paths or register files.

The set of FU assignments with the minimum *completion time* for this operation is passed on as the algorithm recurses up the DFG, and is used as the set of potential consumers for the operation's predecessors. When the algorithm returns down the DFG, the operation is assigned to a specific FU, which is marked busy at that time. In the case of a tie where multiple FU assignments have the same *completion time*, one is chosen arbitrarily.

The example in Figure 5 illustrates the BUG FU assignment algorithm. Beginning at operation h, the algorithm assumes that either of the compatible FUs $+_2$ and $+_4$ will result in a good schedule. Then, it recurses upward on the left-hand path in the DFG and estimates that operation d would complete sooner on FU $+_2$, since it is an ADD operation and its consumer is an ADD operation, and there is a bypass path from $+_2$ to itself. This continues up the DFG with $+_2$ also being chosen for operations b and c. Operation a is valid on FUs M_1 and M_3, both of which have the same *completion time* of 4 (1 cycle delay and 3 cycles to transmit results to operation b via the register file), so assume it chooses M_1 arbitrarily.

On the right-hand path, the algorithm performs in a similar fashion and assigns the operations to FUs M_3 and $+_4$ alternately. Finally, operation h is assigned to $+_2$. This set of FU assignments is passed on as hints to the actual scheduler, and the final schedule is given in Table 1(b), taking a total of 8 cycles.

Integer Linear Programming Formulation. Another method of generating FU prioritization hints is through ILP. Consider the dataflow graph $DFG = (V, E)$ of a compilation region (a basic block or a hyperblock) where V is the set of operations in the region and E the set of edges. An edge e_{ij} indicates a flow dependence from operation v_i to operation v_j. Let R be the set of resources in the machine. For each edge e_{ij}, we introduce a set of variables x_{ij}^{pq}, $p \in \{r_1 | r_1 \in R, r_1$ can execute operation $v_i\}$, $q \in \{r_2 | r_2 \in R, r_2$ can execute operation $v_j\}$, each of which can take a value of 0 or 1. $x_{ij}^{pq} = 1$ indicates that the operation v_i must have priority to be executed on resource p and operation v_j on resource q. Thus, we get constraint 1 (equation on next page) for each edge e_{ij}.

Now, consider the edges e_{ij} and e_{jk}. For each resource q that operation v_j can execute on, we introduce constraint 2. This ensures that the resource hint for operation v_j is compatible across edges e_{ij} and e_{jk}. Also, for a resource pair (p, q), we have a latency l_{pq} which depends on the resource p's latency and whether or not a bypass exists between the resources p and q. For a machine with full bypass interconnect, l_{pq} is just equal to the resource p's latency. For some assignment of values to the variables x_{ij}^{pq}, the latency of the edge e_{ij} is equal to $\sum_{p,q} l_{pq} \times x_{ij}^{pq}$.

$$\sum_{p,q} x_{ij}^{pq} = 1 \quad (1) \qquad \sum_{p} x_{ij}^{pq} + \sum_{\substack{r,s \\ r \neq q}} x_{jk}^{rs} = 1 \quad (2) \qquad \sum_{\substack{p,q,i,j \\ v_j \in V_c}} l_{pq} \times x_{ij}^{pq} \quad (3)$$

Note that any solution to the set of equations 1 and 2 gives a valid set of hints for all operations in V. Now, consider the set of operations $V_c \subseteq V$ which have a slack of 0. These are the operations in the critical path of the DFG. We get a good schedule when the latency on each edge e_{ij}, $v_j \in V_c$ is minimized. Therefore we formulate the problem of FU prioritization hint generation as "minimize equation 3, subject to constraints in equations 1 and 2."

Table 1(c) shows the critical path $a-b-c-d-h$ from Figure 5 mapped to FUs M_3, $+_4$, $+_2$, $+_2$, and $+_2$ respectively, which is an optimal assignment. However, a shortcoming of this formulation is that the other operations have a slack of 1 and do not show up in the objective function of the linear program. Therefore the ILP formulation gives a schedule length of 10 instead of the optimal value of 8.

5 Experimental Results

Methodology. Our system was implemented using the Trimaran toolset [20], a retargetable compiler framework for VLIW/EPIC processors. We ran our experiments on the MediaBench [14] benchmark suite. Two machine models were used: a narrow issue machine consisting of 2 integer, 1 floating point, 1 memory and 1 branch units; and a wide issue machine consisting of 4 integer, 2 floating point, 2 memory and 1 branch units. This specifies the issue width of the machine although the actual machine has a more fine-grained mix of FUs (adders, shifters, multipliers, etc.). For example, the narrow machine has 2 adders, 2 shifters, and 2 multipliers (all integer). The FU latencies are similar to those of the Intel Itanium. We assume a perfect memory system with a 2-cycle latency for loads.

For our experiments, we use a simple cost model to evaluate the cost of the multiplexors and selector logic in the bypass network. The cost is measured in terms of the number of 2-input NAND gates. Specifically, we cost the number of gates in each MUX per source port per FU. The number of inputs to the MUX at a particular FU port is the total number of bypass paths to that port plus an operand wire from the register file. If the number of bypasses is zero, we do not cost the MUX. Each bypass path is scaled by the width of the FU. The bypass selector logic consists of k comparators, for a k-input MUX. All machine models have 64-entry register files (integer, floating point, and predicate). So each comparator compares two 6-bit quantities: the destination register identifier from the bypass path and the source register identifier of the instruction. We ignore the area costs related to routing of the result buses and the delay in cycle time resulting from the complex bypass wires and control logic, since much of this is dependent on the layout of the circuit. Although this is a simplistic cost model, it helps in studying the first order effects of various architectural decisions on the final processor. An approximate cost model will help the architect to study the trends in bypass costs with respect to performance and prune away non-optimal designs.

Results. Figures 6(a) and (b) show two Pareto graphs for explored design spaces: djpeg on the 2111-configuration machine and g721decode on the 4221-configuration machine. The fully bypassed machine has a relative cycle count of 1.0. The rightmost points on both graphs indicate the cost of the initial machine with all non-utilized bypasses removed. While in the 2111-machine, the BUG FU prioritization algorithm was able to better prune the bypasses than ILP, the opposite was true in the 4221-machine. Similarly, varying results appeared across all the benchmarks. Of interest is the "elbow" of the Pareto, where the explorer could not find any more bypasses to remove without a dramatic drop in performance. In the 2111-machine this occurs at a cost of approximately 4500 gates, and in the 4221-machine it is at 12000 gates. The runtime of the bypass customization system is reasonable and ranges from about 1 hour to 10 hours for each benchmark on a 2-GHz Pentium 4.

Figure 6(c) shows, for each benchmark, how much cost savings can be obtained while achieving a given level of performance. On average, one can achieve 90% of the performance of a fully bypassed machine with only 30% of the bypass cost. Some interesting points to note are the first bars of each benchmark, which show how much cost savings can be achieved from the bypass paths without any loss in performance. In some cases, e.g. cjpeg, only 51% of the bypass cost is actually needed to match the performance of a fully bypassed machine. In others, such as g721encode/decode, removing just a single utilized bypass path results in some performance degradation. On the other end of the spectrum, note that on average, 70% of the original performance is realized with a cost of only 11% of the bypasses in the initial machine,.

Figure 6(d) compares the BUG and ILP FU prioritization algorithms. The machine shown has 75% of the utilized bypass paths removed; the baseline is the machine with only the non-utilized bypass paths removed. Since the objective function in the ILP formulation ignores operations not in the critical path, it performs poorer than the BUG algorithm for many benchmarks. On average, BUG performs 6% better than the ILP algorithm.

Figure 7 demonstrates that different applications have very different needs with regards to the bypass network. For each benchmark, the vertical axis represents the bypass paths that were utilized in the machine. Points on the figure represent bypasses that remained after pruning a full bypass machine until performance dropped 15%. Looking at vertical patterns in the graph, we can see that rawdaudio and mpeg2encode need very few bypasses to maintain high performance, since there are relatively few points in their columns. Conversely, programs like djpeg and mesa need a relatively large number of bypasses to maintain high performance. Looking horizontally at the graph, we can tell whether separate applications use the same types of bypass paths. For example, pegwit encode and pegwit decode have high degree of overlap, which suggests that a bypass network customized for one application would work well for both. On the other hand, rasta and rawdaudio have much less overlap, so a customized processor for rawdaudio would run rasta slowly, and vice-versa.

In general, the most utilized bypasses were those from ALU to ALU; ALU to/from memory units; and ALU to branch units. Several benchmarks such as rawcaudio and mpeg2encode used few bypasses to and from the complex integer units (multiplier, divider), indicating that those benchmarks do not have many of those operations on their critical paths. As expected, only those benchmarks using floating-point code had bypasses to and from the floating-point unit. The most important observation is that application demands on the bypass network vary greatly, even among applications from MediaBench which are all in the same domain.

6 Conclusion

In this paper, we have shown that the vast majority of the bypass network is either never or rarely utilized for a given application. We have developed algorithms to efficiently explore the design space of bypass networks and to compile to machines with incomplete bypass. Our results show that on average we can remove 70% of non-zero utilized bypasses and still maintain 90% of the original performance.

References

[1] A. Abnous and N. Bagherzadeh. Pipelining and Bypassing in a VLIW processor. *IEEE Transactions on Parallel and Distributed Systems*, 5(6), June 1995.

[2] S. Aditya, V. Kathail, and B. R. Rau. Elcor's Machine Description System: Version 3.0. Technical Report HPL-98-128, Hewlett-Packard Laboratories, Oct. 1998.

[3] S. Aditya, B. R. Rau, and V. Kathail. Automatic Architecture Synthesis and Compiler Retargeting for VLIW and EPIC Processors. In *Proc. of ISSS*, Nov. 1999.

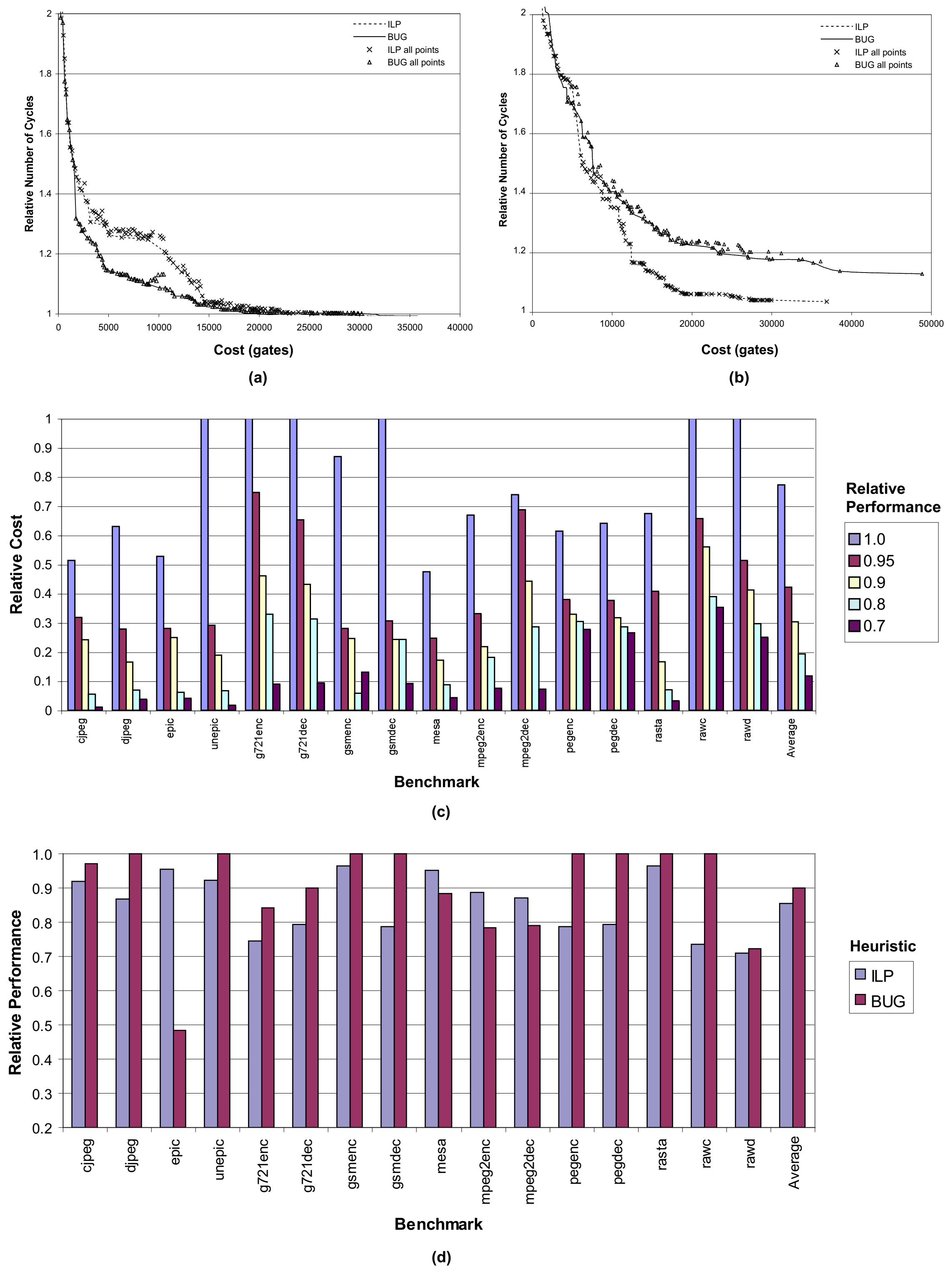


Figure 6. (a) Pareto for djpeg on 2111 machine; (b) pareto for g721decode on 4221 machine; (c) cost of bypasses required to achieve given performance levels; (d) comparison of BUG and ILP prioritization heuristics on a machine with 25%-cost bypass paths.

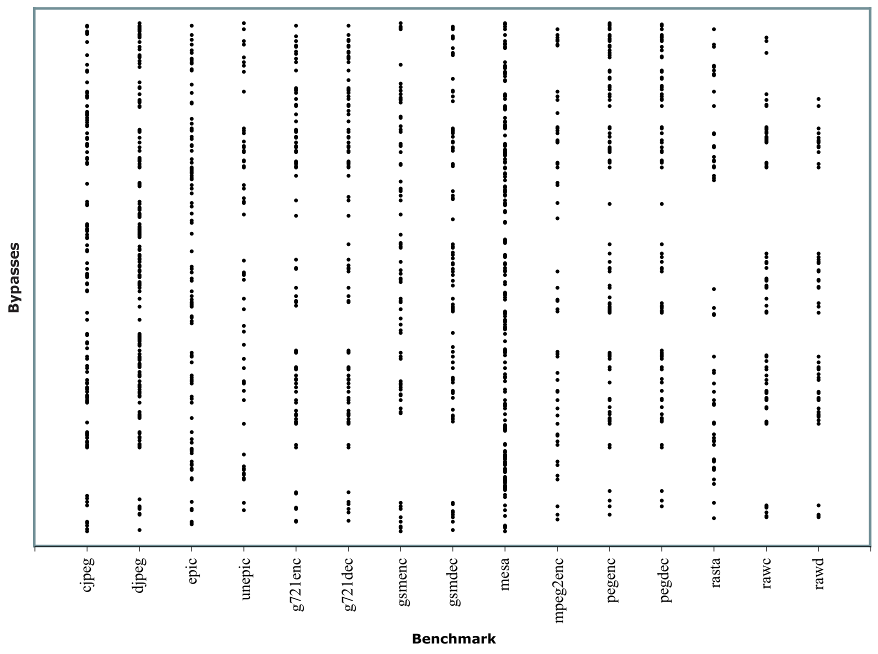


Figure 7. The bypasses used by each application at the 85% performance threshold.

[4] P. Ahuja, D. Clark, and A. Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proc. of Micro-28*, Dec. 1995.

[5] E. Bloch. The Engineering Design of the Stretch Computer. In *Proc. of the Easter Joint Computer Conf.*, 1959.

[6] M. D. Brown and Y. Patt. Using Internal Redundant Representations and Limited Bypass to Support Pipelined Adders and Register Files. In *Proc. of HPCA-8*, Feb 2001.

[7] M. Buss, R. Azevedo, P. Centoducatte, and G. Araujo. Tailoring Pipeline Bypassing and Functional Unit Mapping to Application in Clustered VLIW Architectures. In *Proc. of CASES*, Nov. 2001.

[8] R. Cohn, T. Gross, M. Lam, and P. Tseng. Architecture and Compiler Tradeoffs for a Long Instrction Word Microprocessor. In *Proc. of ASPLOS-3*, April 1989.

[9] J. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1985.

[10] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-Fit Processors: Letting Applications Define Architectures. In *Proc. of Micro-29*, Dec 1996.

[11] G. Hekstra et al. TriMedia CPU64 Design Space Exploration. In *Proc. of ICCD*, Oct. 1999.

[12] I. Karkowski and H. Corporaal. Design Space Exploration Algorithm For Heterogeneous Multi-processor Embedded System Design. In *Proc. of DAC-35*, June 1998.

[13] V. Lapinskii. *Algorithms for Compiler-Assisted Design Space Exploration of Clustered VLIW ASIP Datapaths.* PhD thesis, University of Texas, 2001.

[14] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of Micro-30*, Dec. 1997.

[15] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors. In *Proc. of DAC-21*, 1984.

[16] E. McLellan. The Alpha AXP Architcture and 21064 processor. *IEEE Micro*, 13(3):36–47, June 1993.

[17] S. Palacharla. *Complexity-Effective Superscalar Processors.* PhD thesis, Wisconsin, Madison, 1998.

[18] J. Sato, M. Imai, T. Hakata, A. Alomary, and N. Hikichi. An Integrated Design Environment for Application Specific Integrated Processors. In *Proc. of ICCD*, Oct. 1991.

[19] G. Snider. Spacewalker: Automated Design Space Exploration for Embedded Computer Systems. Technical Report HPL-2001-220, Hewlett-Packard Laboratories, Sept. 2001.

[20] Trimaran. An Infrastructure for Research in ILP. http://www.trimaran.org.

[21] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A Fast Flexible Architecture for Secure Communication. In *Proc. of ISCA-28*, June 2001.