Custom–Fit Processors: Letting Applications Define Architectures

Joseph A. Fisher, Paolo Faraboschi and Giuseppe Desoli

Hewlett-Packard Laboratories Cambridge
1 Main Street, Cambridge, MA 02142
{jfisher,frb,desoli}@hpl.hp.com

Abstract

In this paper we report on a system which automatically designs realistic VLIW architectures highly optimized for one given application (the input for this system), while running all other code correctly. The system uses a product-quality compiler that generates very aggressive VLIW code. We retarget the compiler until we have found a VLIW architecture idealized for the application on the basis of performance, a cost function and a hardware budget.

We show that we can automatically select architectures that achieve large speedups on color and image processing codes. Specialization is shown to be very valuable: The differences between architectural choices, even among reasonable-seeming architectures having similar costs, can be very great, often a factor of 5 (and sometimes much more). We show also that specialization is also very dangerous. A reasonable choice of architecture to fit one algorithm can be a very poor choice for another, even in the same domain. There is sometimes an architecture, near in cost and performance to the best, that does much better on a second algorithm.

1. Introduction: Custom-Fit Processors

Many people had the same reaction upon first becoming aware of VLSI microprocessors: can we somehow design these chips by merely writing a program that describes what they are to do? This is the "silicon compiling" problem in its full glory. Despite amazing progress in areas such as compiling control circuits using a "sea of gates" approach, and despite tremendous progress in the automation of many of the steps of design and production, we are very far from taking a functional description of a microprocessor and automatically producing a silicon layout.

Although silicon compiling is well beyond us, in this paper we consider an even more ambitious problem: rather than generating a microprocessor automatically from a high-level description, we would like to generate it from the applications it will run. In particular, given an embedded processor running on an "appliance" product, we call the general-purpose processor which is designed to scream on the embedded application a "Custom-Fit Processor". The process we use to derive this processor design falls into the general class of technologies referred to as Hardware/Software Codesign [18, 15, 16, 20].

Generating Custom-Fit Processors automatically is a superset of generating general-purpose processors, and is thus strictly harder still. When faced with such an overwhelmingly impossible task, there are two approaches generally taken:

- One can build a very small thing (e.g. synthesizing a 4-bit adder) and hope to learn while approaching reality from below.
- One can build a "toy", filled with unreality, and then try to make it successively more realistic, approaching reality from above.

Here we suggest a third approach, and our research program is dedicated to carrying this out:

- One can restrict the problem one is attacking, but then, within that restricted framework, do something that is completely realistic and is an end-to-end solution. From that base, move towards a less-restricted solution. In a sense, this starts out being realistic, but approaches greater generality "from the side".

The restriction we impose is the following: We design a VLIW [11] architecture in which virtually all characteristics can be changed: memory sizes and hierarchy, register sizes and ports, the "cluster" structure of the architecture, the kinds of functional units and their repertoires, the latencies of the functional units, and the connection and communication topologies of all of these. At the same time,

1072-4451/96 $5.00 © 1996 IEEE

the code transformations that are done as part of the hardware/software codesign process are applied. This adds to that mix the limitations of what the compiler is capable of. Any codesign system will have this latter restriction, almost always much more than we do. Unfortunately, this is almost never acknowledged in research results.

Within this framework our methodology is easy to describe, and similar to what has been described elsewhere (for example, in [14, 12, 13, 2, 8, 19, 3, 6, 7]). For example De Gloria and Faraboschi [8] carry out almost exactly this framework, but using tools which are much less mature.

This framework is particularly interesting right now, because it is now practical to put enough millions of transistors on an inexpensive die to make a very powerful and general VLIW – witness the media processors now appearing from several vendors [9, 21, 4, 10].

1.1. This Investigation

We believe the work described below is unique in the following sense:

- We are doing it with a productized, ambitious compiler that exposes and schedules a lot of ILP. Previous studies have been done in environments in which very small percentage differences have been available; at best they have found small factor speedups. Instead, we find very large factor speedups, even between relatively similar cost, reasonable-seeming architectures.

The experiments done here are a characterization of the effectiveness of tailoring ILP hardware to given applications. We are attempting to shed light on the following broad questions:

- What is the performance of custom hardware at a given cost, when compared to more general hardware at that same cost?
- How does the hardware you would build differ for different sections of code in similar application areas? How does it differ from hardware built for several routines at the same time?
- How effective are search methods aimed at finding the appropriate architecture?

2. Experimental Methodology and Infrastructure

2.1. The C Compiler

Our main tool in this investigation is the *Hewlett-Packard Laboratories Cambridge C Compiler*. It is a direct descendant of the Multiflow compiler, which has been reported upon in detail elsewhere, particularly in [17]. For our purposes, it has the following qualities:

- It is a productized, real-world, highly-optimizing compiler.
- It generates ILP code as aggressively as any compiler we have ever heard of; we think more than any other compiler ever built.
- It generates code from table-driven architectural descriptions in the following sense: if you have a description of an architecture for which you are generating good code, you can change most of the "normal" architectural parameters to produce a new model, and continue to generate good code.

We thus are able to use it to explore a design space of architectures to fit a processor to a given application.

2.2. Searching For A VLIW Architecture

Our basic experimental method involves the following loop:

- Using some search method, search for a new candidate architecture
- Measure the cost of the architecture
- Build a version of our compiler that generates good code for that architecture
- Generate the code
- Measure the goodness of the code
- Repeat until satisfied

In the past, many researchers have implemented similar loops. They have typically concentrated upon search techniques, or upon the selection of special-purpose functional units to match the functionality needed in a loop. Our philosophy here is different. Following the RISC/VLIW religion, we want to build simple hardware that does the basic, simple operations, but uses lots of ILP to get a speedup. So we try to match the *structures* and *sizes* of the architecture to the application, rather than specific opcodes.

Similarly, it is likely that search techniques to prune the space of architectures under consideration would be very successful. Here, instead, we searched exhaustively through a huge space: despite being real-world tools, our tools are fast enough and computers are now sufficiently fast enough to make this practical. We are confident that any good search technique could cut down significantly on our processing time (see Table 3) without greatly affecting the results reported upon here.

2.3. Benchmarks

We were interested in measuring how different an application-specific microprocessor would be when tailored for different tasks within a single application domain. These seem like relevant questions: right now people build chips to do specifically one subtask of an application, when a general-purpose processor is not sufficient (e.g. for MPEG video compression/decompression); additionally, we now have media processors, which are specialized for an application area, but not a single subtask. This makes intuitive sense, as the subtasks in a single application area often seem to have similar compute structures, as is the case in media processing.

Benchmark	Description
A	FIR symmetrical filter implemented using a 7x7 convolution kernel.
C	Inverse DCT transform with dequantization of the DCT coefficients. The algorithm used is the Arai, Agui and Nakjima algorithm for scaled FDCT/IDCT, with some improvements, as described in [1, 22].
D,E	Color conversion from the RGB to the YCbCr color space (and vice versa, as described in the JPEG standard)
F	Halftoning via standard Floyd-Steinberg error diffusion (no stochastic weights update). The benchmark produces triplets containing 1 bit halftoned pixels.
G	1D bilinear scaling by integral factors along columns.
H	3x3 median filter using the standard algorithms, not using a "smart" version of the median.

Table 1. The individual benchmarks.

We picked color output routines, which are quite easily available in the public domain, and are quite similar to those used in many media-processing applications. These routines often contain a large quantity of potential ILP. All the benchmarks except C have as input a row of a full color RGB image. We have converted all floating point to fixed, as is common in this kind of processing. Proper source code transformations have been applied to all benchmarks to expose ILP (loop transformations, if-conversion, etc.). These same transformations speed the code up on virtually all superscalar and VLIW architectures and implementations. Table 1 describes the benchmarks we used.

We wanted to know how architectures which were optimized for those individual routines would compare with architectures optimized for collections of the routines. Thus

Benchmark	Description
GF	1D bilinear scaling followed by Floyd-Steinberg halftoning.
GEF	1D bilinear scaling followed by E, a YUV$\rightarrow$RGB color space conversion, followed by Floyd-Steinberg halftoning.
DH	RGB$\rightarrow$YUV color space conversion followed by a 3x3 median filter.
DHEF	RGB$\rightarrow$YUV color space conversion followed by a 3x3 median filter, followed by E, a YUV$\rightarrow$RGB color space conversion, followed by Floyd-Steinberg halftoning.

Table 2. The jammed benchmarks.

we also ran combinations of the above, jammed into single loops, avoiding the intermediate memory store/load otherwise needed. Table 2 shows a description of the "jammed" benchmarks.

For example, figure 1 shows the D (Floyd-Steinberg error diffusion) benchmark in C, implemented in the standard form found in image processing literature.

2.4. Running the Experiment

The experiment was set up in such a way that we were rebuilding a compiler for each architecture and then running the compilation for all benchmarks and for different unrolling factors. When the compiler started spilling register contents for a given unrolling, we stopped considering that unrolling factor and all larger ones.

The performance of clustered architectures (see Section 3.1) was not computed for all possible combinations, to avoid an exponential explosion of runtime and data. To account for clustering, we computed a "correction value" as a function of the number of clusters, by running a set of separate experiments for a few significant architecture data points in the defined space. In our experience, this approximation is enough to account for the effects of clustering.

For this experiment, we ran 5730 compilations of the benchmarks, on 191 architectures (plus their associated clustering values). The time to re-compile a customized compiler was relatively short (about 50 seconds), since only the machine model needed to be re-linked into the executable.

The time to run a single compilation benchmark varied significantly between a couple of seconds and a few minutes. On average, it took on the order of 28 seconds per benchmark, adding up to about 48 hours of running time for the whole experiment. The platform used for the experiment was a 9000/770 HP workstation, 100MHz clock, 256MB main memory.

```
FSDline ( ubyte * linein,
          ubyte * lineout,
          int plane_size )
{
  int i, color;
  int16 *ep, Err[3], errTemp[3];
  int16 errTempOff[3], oldErr[3];
  ubyte *dp, *op, out[3], bitmask;

  dp = linein;
  ep = errBuf + 3;
  op = lineout;
  Err[0] = Err[1] = Err[2] = 0;

  errTemp[0] = ep[-3 + 0];
  errTemp[1] = ep[-3 + 1];
  errTemp[2] = ep[-3 + 2];
  bitmask = 0x80;

  for (i = 0; i < plane_size; i++)
  {
    for (color = 0; color < 3; color++)
    {
      errTempOff[color] = errTemp[color];
      errTemp[color] = ep[color];
      oldErr[color] = Err[color];
      Err[color] =
         (errTemp[color] + ((Err[color]*7+8)>>4)+
         ((int) dp[color] << ((2*8)-13)));
      out[color] =
         ((Err[color] > (128 << ((2*8)-13)))
         ? out[color] | bitmask : out[color]);
      Err[color] =
         ((Err[color] > (128 << ((2*8)-13)))
         ? Err[color] - (255 << ((2*8)-13))
         : Err[color]);
      errTempOff[color] += ((Err[color]*3+8)>>4);
      errTemp[color] =
         ((Err[color] * 5 + oldErr[color]+8)>>4);
      ep[-3 + color] = errTempOff[color];
      op[color] = out[color];
    }
    dp += 3, ep += 3;
    if (bitmask == 0)
    {
      op = op + 3;
      out[0] = out[1] = out[2] = 0;
      bitmask = 0x80;
    }
    else
      bitmask = bitmask >> 1;
  }
}
```

Figure 1. The Floyd-Steinberg algorithm.

Table 3 shows the basic data concerning the computation time of the experiment.

3. Architecture Cost and Performance

Computing the cost and performance of an architecture from a set of high-level parameters (such as number of ALUs, multipliers, registers, ports, etc.) is a nontrivial task. Several implementation choices exist and the trade-off between choices varies widely depending upon available VLSI technology, target application, area and power requirements, design methodology and so on.

In this paper we have simplified the problem and only consider the cost of building the CPU datapath. Other factors, such as pinout requirements and cost of the memory system, are not considered in the cost equations. Considering these factors would affect the numbers we report, but probably would not materially affect our conclusions.

# runs	5730
# architectures	191
runtime per architecture	897s (15 m)
compiler time per benchmark	28s
compiler compile time per arch.	50s
total time	171449s (48 h)

Table 3. Experiment computation time.

We estimate the cost of the datapath in terms of silicon real estate relative to a baseline configuration. The figures that we use are derived from an analysis of existing designs in current VLSI technology. They are certainly not close to exact figures, but we believe are representative enough to support the conclusions of this paper.

3.1. Clusters and Architectural Parameters

Our architecture template (Figure 2) is a multi-cluster machine, composed of (nearly) identical clusters containing functional units and local register banks. The communication between clusters happens across a set of global connections, and is explicitly scheduled by the compiler. The Multiflow Trace [5] follows exactly this structure. The reason for clusters, which are not independent, but rather share a single long instruction, is to avoid register banks with too many ports. Thus instead of a single register bank supporting, say, 8 ALUs, we might split it into 4 register banks, each supporting two ALUs. In order to use an ALU, the operands it requires must be in the associated register bank, or must be moved there with an explicit move in a prior instruction. The fact that the different clusters are (nearly) identical makes the chip easier to fabricate as well. The cluster differ in that the single branch unit resides on cluster #0 and is not duplicated in the others.

In addition to that, we consider a multi-level memory system composed of a *Level 1 Memory* and a *Level 2 Memory*. *Level 1* Memory is used to model the global memory of the system, and has a fixed throuput for all the experiments. *Level 2* Memory varies in terms of number of parallel accesses and latency.

Table 4 explains the parameters we take into account. Some of the settings we used were completely determined by our choice of the initial parameters. These include register file ports, communication paths and cycle speed. Table 5 shows a description of these derived parameters.

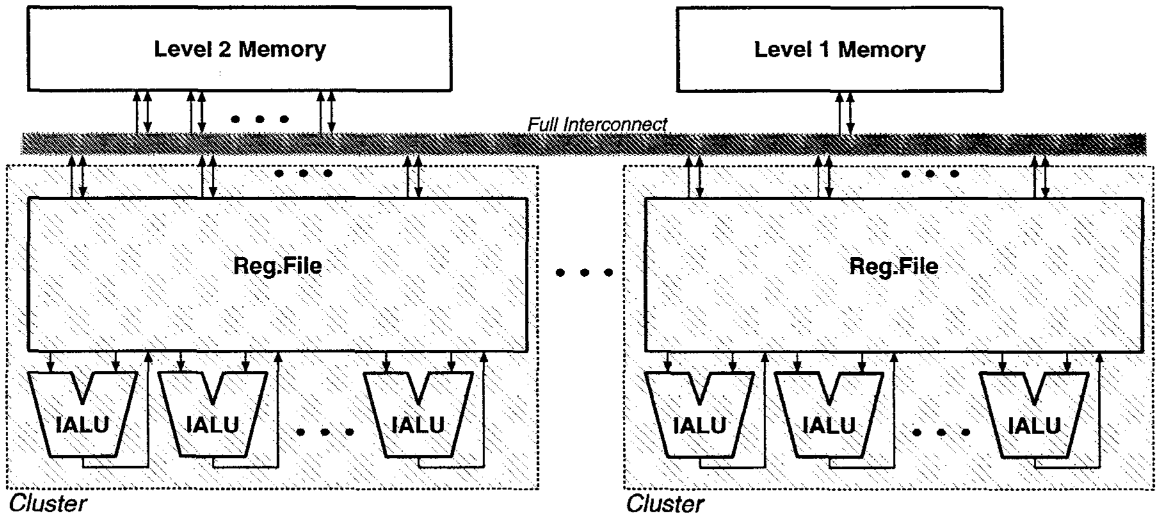

Figure 2. The architecture template

Parameter	Description
Clusters	Ranges between 1 and 16.
IALUs	Ranges between 1 and 16. All operations have latency of 1 cycle, except multiplications (2 cycles, pipelined).
ALU Repertoire	We eliminated floating point units by hand before we started. These routines do little floating point, and the cost function would have eliminated it in any case, so we saved the trouble. Among the integer units, the only choice presented in this experiment is whether or not a given ALU is capable of integer multiply. We allowed between 1/4 and 1/2 the ALUs to be IMULs, however at least 1 IMUL was always present. This methodology allows us to give any opcode choice to the compiler. We limited this experiment for expository reasons, and because our philosophy in general is to design an architecture from building blocks rather than synthesizing lower-level special- purpose hardware.
Register Sizes	We allowed between 64 and 512 registers total (for all clusters).
Memory System	We picked many different configurations for this experiment, but found that considering all of them muddied the insights available in this paper, but did not change the results. We thus decided to limit the exposition to only a few choices: always a single *Level 1 Memory* port and between 1 and 4 accesses to *Level 2 Memory*. The latency of an access to *Level 1 Memory* is always 3 cycles (non-pipelined), while the latency to *Level 2 Memory* varies from 2 to 8 cycles (non-pipelined).

Table 4. The architecture parameters.

Parameter	Description
Register Ports	We varied these with the requirements of the other functional units. In a full system, it is useful to consider this an independent variable, since it greatly affects the cost of the system.
Connectivity	As in the number of register ports, we varied the connectivity according to the needs of the functional units, but it could have been allowed to be a more general parameter (and in the production of a chip you will build, will be).
Cycle Speed	We used an approximation to quantify the effect of cycle speed of our architectural choices. This is how we believe it best to treat cycle speed in a system like this, though sometimes it may be considered an independent parameter, as in making silicon technology choices, or in area vs. speed tradeoffs in designing functional blocks.

Table 5. The derived parameter settings.

3.2. The "Baseline" System

We used as our baseline system one with 1 ALU, which could do IMUL, 1 reference to *Level 1* Memory and 1 to *Level 2* Memory (8 cycle latency), and 64 registers, all in 1 cluster. Our costs and performance models, explained in the following sections, are scaled to make this system cost 1 unit.

Note that this system is capable of a great deal of ILP, due to its multiple issue capability, and its pipelining.

3.3. Computing Architecture Cost

The cost function for an architecture is computed as follows:

$$COST = c \cdot X_{dp}(p) \cdot (Y_{reg}(r,p) + Y_{alu}(a) + Y_{mul}(m))$$

- c is the number of clusters
- a is the number of ALUs per cluster
- m is the number of ALUs per cluster able to do integer MULs
- r is the number of registers per cluster
- l is the number of memory accesses per cluster
- p is the number of ports of the register file in a cluster, computed as a function of the ALUs (a), and the memory ports (l): $p(a,l) = (3 \cdot a) + (2 \cdot l)$
- $X_{dp}(r,p)$ is the datapath width of a cluster, computed as: $X_{dp}(p) = k_1 \cdot p$
- $Y_{reg}(r,p)$ is the register file height for a cluster, computed as: $Y_{reg}(r,p) = r \cdot (k_2 \cdot p + k_3)$
- $Y_{alu}(a)$ is the height of the ALUs for a cluster, computed as: $Y_{alu}(a) = k_4 \cdot a$
- $Y_{mul}(m)$ is the height of the MULs for a cluster, computed as: $Y_{mul}(m) = k_5 \cdot m$
- $k_1 \cdots k_5$ in the above equations are fitting parameters computed from observation of existing designs.

The costs range from 1.0 (for the baseline) to about 100 for the most ambitious architectures (16 ALUs, 8 MULs, 512 registers, 4 memory ports, 1 cluster). For example, Table 6 shows the cost of some of the architectures that we have considered in our experiments.

These numbers are certainly approximate, but we believe they are realistic enough to allow one to generalize from the results of this study.

3.4. Cycle Speed

The complexity of an architecture impacts the cycle time, a factor we must take into account in any realistic evaluation. In this experiment, we have tried to come up with a reasonable derating factor (vs. the baseline architecture) that applies a cycle time increase as a function of the register file ports. The underlying assumption is that, as in most designs, the read stage of the pipeline is the limiting factor for cycle speed.

The function that we use assumes a quadratic relationship between cycle time and number of ports, and, for example, Table 7 gives the following values for some of the architectures we considered:

IALU	IMUL	L2MEM	REGS	Clusters	Cost
1	1	1	64	1	1.0
2	1	1	64	1	1.7
4	2	1	128	1	6.5
4	2	1	128	2	3.6
8	4	1	256	1	28.7
8	4	1	256	2	13.1
8	4	1	256	4	7.4
16	8	1	512	1	93.4
16	8	1	512	2	38.4
16	8	1	512	4	19.0
16	8	1	512	8	12.2

Table 6. Examples of the cost of some of the architectures considered in the experiments. Costs are expressed as relative ratios versus the cost of the baseline configuration (the first line of the table). L2MEM is the number of level2 memory ports.

IALU	L2MEM	Clusters	Cycle
1	1	1	1.0
2	1	1	1.1
4	1	1	1.5
4	1	2	1.1
8	1	1	2.7
8	1	2	1.4
8	1	4	1.1
16	1	1	7.3
16	1	2	2.7
16	1	4	1.5
16	1	8	1.1

Table 7. Examples of cycle speed derating factors for different architecture configurations. Again, values are relative to the baseline configuration (the first line of the table).

4. Results

The performance of the benchmarks (*"su"*) and the cost (*"c"*) of the architectures are displayed on Tables 8, 9, 10 and Figures 3, 4.

In our experiments, we describe architectures by means of a n-uple of 6 parameters:

$$(a, m, r, p_2, l_2, c)$$

where

- a is the total number of ALUs
- m is the total number of ALUs capable of executing an integer MUL
- r is the total number of registers
- p_2 is the total number of parallel accesses to *Level* 2 Memory
- l_2 is the latency in cycles of accesses to *Level 2* Memory
- c is the number of clusters

So, for instance, the description *(4 2 256 1 4 4)* (first line of Table 8) identifies an architecture with 4 ALUs (1 per cluster), 2 of them capable of a MUL, 256 registers (64 per cluster), 1 port to *Level* 2 memory with a 4-cycle access latency and 4 clusters.

The basic unit of measure for our experiments is called "Speedup", and it represents the factor performance improvement that the given architecture gets on the benchmark in question, compared to the running time of that benchmark on the baseline system. Note that this is *not* the speedup due to instruction-level parallelism. The baseline system already offers a significant amount of ILP. In this paper we wanted to avoid the "maxpar wars", and just concentrate on the relative evaluation of the choices we had.

4.1. Architectural Performance Vs. Cost

Figure 3 is a scatter diagram that shows the cost and speedup for each of the 191 architectures (after the best cluster arrangement had been selected) for each of the individual algorithm benchmarks (A, C, D, F, G, H), while Figure 4 shows the same for the jammed benchmarks. The line in each scatter diagram is drawn through all of the best cost/performance alternatives for each benchmark.

Note that most of the diagrams contain various performance levels in which several points are approximately in a straight horizontal line. In those cases, the apparent explanation is that the increasing costs as one goes to the right are due to the addition of features to the leftmost architecture that are not helping it achieve much better performance on the given benchmark. Sometimes a feature is very relevant to a benchmark, and then a new plateau is reached with several points in a higher performance level straight line as that feature is added.

4.2. Design For One Algorithm, Run Another

One of the things we most wanted to find out in this experiment was this: when you design for one application or algorithm and run on another, what happens? In particular, one might expect to run into these situations:

1. Perhaps specialization hardly matters at all. Architectures for a given application all perform pretty much the same, with small percentage differences among them, as long as one considers only "reasonable" architectures. Or,
2. Perhaps architectures differ a lot, but there is some independent measure of goodness for them that does not vary much with the benchmarks (except in the grossest of choices, such as whether to include floating point hardware). Architectures built for applications in a narrow domain are more-or-less "well ordered" That is, for any two architectures, the same one is virtually always better than the other across the applications. Or,
3. Perhaps the architectures optimized for different algorithms differ a lot, and how good they are heavily depends upon which algorithm was used to guide the choice.

To answer these questions in the context of the experiment done here, we decided to set up several situations in which a designer might choose an architecture. We allowed the following two degrees of freedom:

- An upper bound on the cost of the architecture. We arbitrarily chose costs of 5, 10, and 15 as constraints a designer might be working with. We call these low, medium, and high cost architectures, respectively.
- The degree to which the designer would be willing to select an architecture that was not the absolute best for the algorithms for which the architecture was optimized, in order to perform better on the other applications.

In particular, in Tables 8, 9, 10 we considered, for each algorithm, the architecture that would do best without exceeding the COST parameter.

When the RANGE parameter was 0, the architecture was chosen without considering the other applications at all, and the table (in this case the *(Range=0%)* portion of the Tables) shows, in the rows across, how well that architecture performed on each application. However, what if the hardware designer wanted to optimize for a given algorithm, but knew something about the domain and was willing to factor in, at least somewhat, other algorithms? In the middle portion of the Tables we show what happens when the designer does the following: For each algorithm, attempt to pick the best

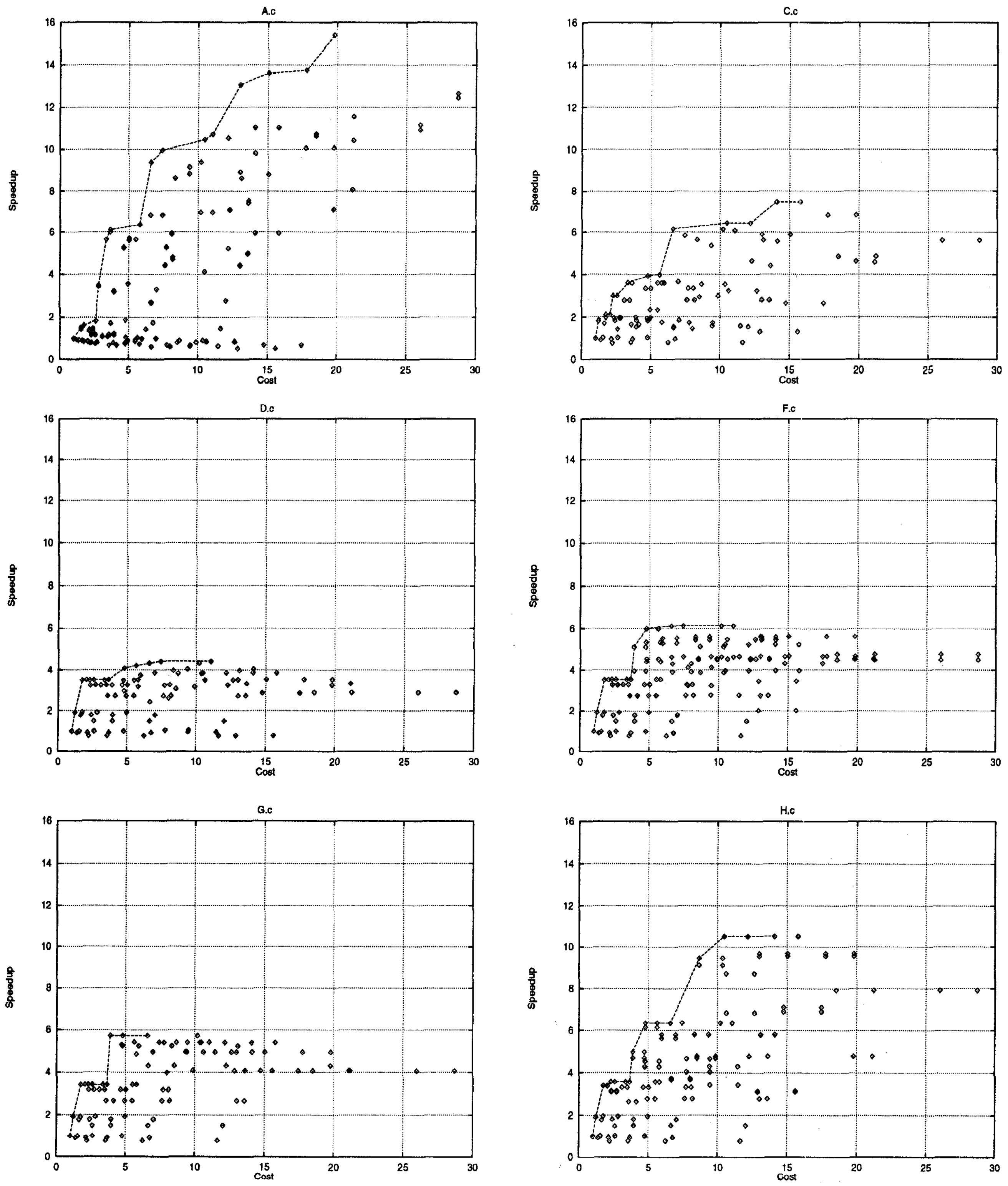

Figure 3. Cost/Speedup scatter diagrams for the original benchmarks

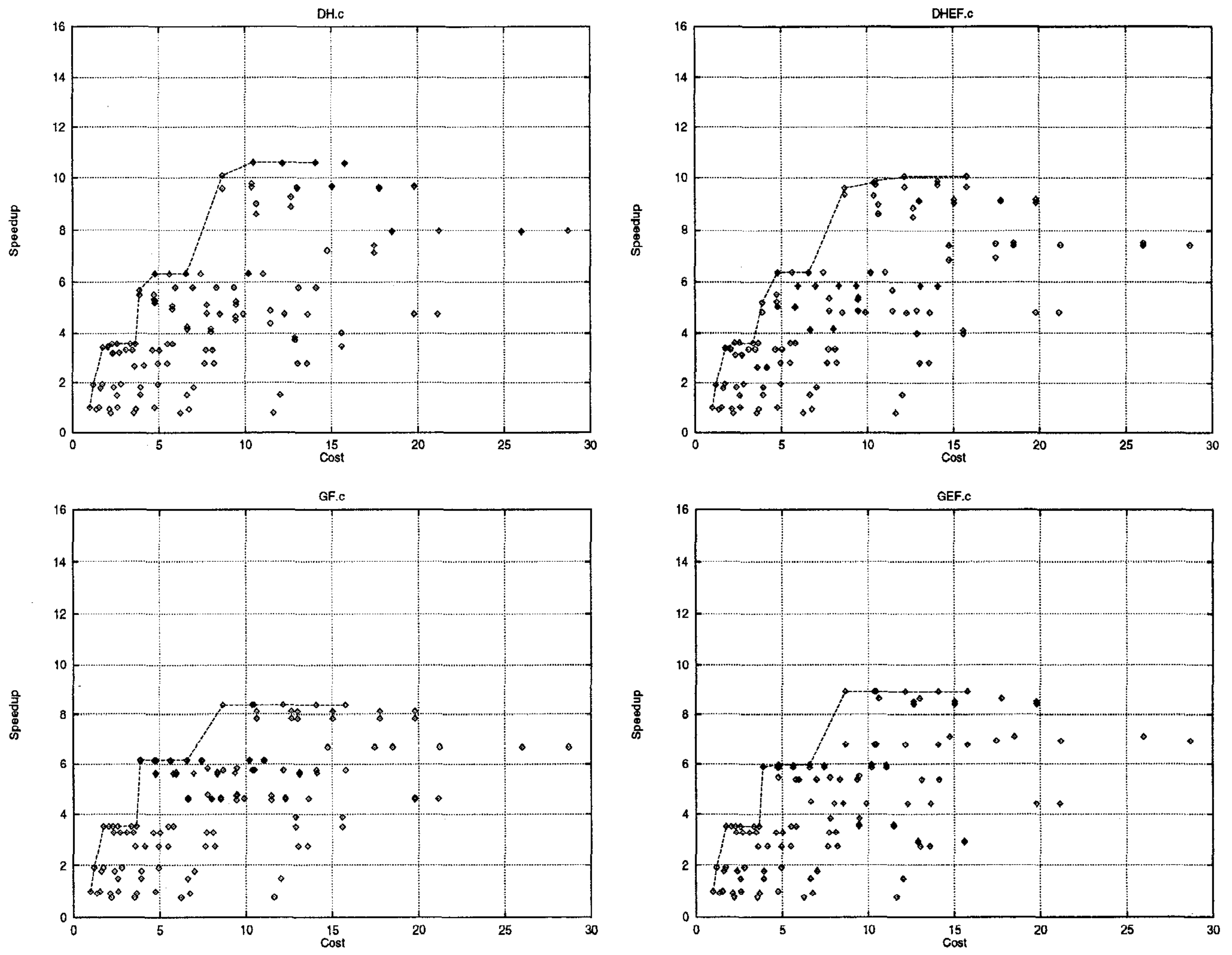

Figure 4. Cost/Speedup scatter diagrams for the jammed benchmarks

architecture, but be willing to back off by as much as 10% of performance (the RANGE parameter) in order to make the average of the other applications better. For the medium cost models, in fact, it is also instructive to see what happens when the designer is willing to give up as much as 50% of the performance to make the others better.

For example, in the medium cost tables, the architecture chosen as the best for algorithm GEF executed that application with a speedup over the baseline of 8.93. But when the RANGE is allowed to go to 50%, the selection mechanism decides to give up 8 of its ALUs, and with that budget buy back another 128 registers (and make some other small changes). This has the effect of lowering the GEF speedup over the baseline from 8.93 down to 5.97. However, in so doing, it makes the overall performance go from an average of 3.9 up to 5.8, and it makes A go from a pathological .89 speedup to 6.82. (Note that in the original model chosen for GEF, there were too many ALUs and too few registers for an algorithm like A. The compiler gets greedy and gets into trouble. This is a known problem, and one hard to avoid with schedulers of this nature. The revised architecture choice gets back to a more balanced system).

Finally, we looked at the question of which architecture minimized the total running time of all the applications at a given cost. Since we used the same mechanism as the above (but with the RANGE set to infinity), we reproduced the result as the same sort of table. However, in that case all the architectures make the same decision, so we only show a single line, at the bottom of each table, giving the performance of the architecture on the given application.

Again following the medium cost model picking an architecture for GEF, the picker gives back just a little more performance than it had at the 50% RANGE (by making changes that are "at the noise level") but improves the overall average from 5.8 to 6.1.

Cost=5.0 Range=0%											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
A(4 2 256 1 4 4)	(3.7 3.6)	$\boxed{6.12}$	3.60	3.52	3.54	3.43	3.58	3.53	3.52	3.56	3.60
C(8 2 128 1 4 4)	(3.8 4.8)	1.05	$\boxed{3.93}$	4.09	6.00	5.72	6.35	6.16	5.86	6.33	6.38
D(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	$\boxed{4.09}$	6.00	5.72	6.35	6.16	5.86	6.33	6.38
F(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	$\boxed{6.00}$	5.72	6.35	6.16	5.86	6.33	6.38
G(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	$\boxed{5.72}$	6.35	6.16	5.86	6.33	6.38
H(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	$\boxed{6.35}$	6.16	5.86	6.33	6.38
GF(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	6.35	$\boxed{6.16}$	5.86	6.33	6.38
GEF(8 2 128 1 8 4)	(3.8 4.8)	1.04	3.93	4.09	4.53	5.72	6.15	6.14	$\boxed{5.97}$	6.31	6.36
DH(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	$\boxed{6.33}$	6.38
DHEF(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	6.33	$\boxed{6.38}$

Cost=5.0 Range=10%											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
A(4 2 256 1 4 4)	(3.7 3.6)	$\boxed{6.12}$	3.60	3.52	3.54	3.43	3.58	3.53	3.52	3.56	3.60
C(8 2 128 1 4 4)	(3.8 4.8)	1.05	$\boxed{3.93}$	4.09	6.00	5.72	6.35	6.16	5.86	6.33	6.38
D(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	$\boxed{4.09}$	6.00	5.72	6.35	6.16	5.86	6.33	6.38
F(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	$\boxed{6.00}$	5.72	6.35	6.16	5.86	6.33	6.38
G(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	$\boxed{5.72}$	6.35	6.16	5.86	6.33	6.38
H(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	$\boxed{6.35}$	6.16	5.86	6.33	6.38
GF(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	6.35	$\boxed{6.16}$	5.86	6.33	6.38
GEF(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	6.35	6.16	$\boxed{5.86}$	6.33	6.38
DH(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	$\boxed{6.33}$	6.38
DHEF(8 2 128 1 4 4)	(3.8 4.8)	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	6.33	$\boxed{6.38}$

Cost=5.0 Range=∞(=490%)											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
all(8 2 128 1 4 4)	(3.8 4.8)	$\boxed{1.05}$	$\boxed{3.93}$	$\boxed{4.09}$	$\boxed{6.00}$	$\boxed{5.72}$	$\boxed{6.35}$	$\boxed{6.16}$	$\boxed{5.86}$	$\boxed{6.33}$	$\boxed{6.38}$

Table 8. Speedup results for low cost (< 5.0) architectures.

5. Conclusions

From the above examples, we can see that, when using a real-world compiler and code that contains a lot of ILP, the architecture choices we make are quite sensitive to the application being tailored to.

By allowing the designer a little freedom to pick less than the absolute best implementation for the target applications, we can often make dramatic improvements in how that implementation will process other applications. This certainly flies in the face of the concept of "Custom-Fit Processing", especially when we see dramatic losses in performance of the original target algorithm that are made to satisfy the overall picture. This can be seen most dramatically in the medium-cost models, in which several of the algorithms run at 60-70% of their performance on a more tailored architecture, and on the low-cost model, in which one application gets into pathologically bad trouble and runs at about 17% of its performance on the architecture made for it.

We believe that, given the maturity and real-world properties of the tools we are using, this is probably a realistic assessment of what designers will face in doing high-level synthesis. If and when the cost of individual chip design becomes very much lower than it is today, it will make a lot of sense to build chips for the narrowest of embedded applications. Today, that seems like a dangerous route to attempt without a very strong apriori knowledge of what will run on the chip.

References

[1] Y. Arai, T. Agui, and M. Nakajima. A fast DCT-SQ scheme for images. *Trans. of the IEICE*, 71(11):1095, Nov. 1988.

[2] M. Auguin, F. Boeri, and C. Carriere. Automatic exploration of VLIW processor architectures from a designer's experience based specification. In *3rd Int. Workshop on Hardware/Software Codesign*, pages 108–115, Sept. 22-23, Grenoble, September 1994.

[3] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, Oregon, December 1–4, 1992. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 23(1–2), December 1992.

[4] B. Case. First TriMedia chip boards PCI bus. *Microprocessor Report*, 9(15):22–28, November 1995.

[5] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. P. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International*

Cost=10.0 Range=0%											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
A(8 4 256 1 4 4)	(6.1 7.4)	$\boxed{9.94}$	5.84	4.42	6.13	5.42	6.35	6.16	5.86	6.31	6.38
C(8 2 256 1 4 4)	(6.1 6.6)	9.38	$\boxed{6.15}$	4.33	6.13	5.72	6.35	6.16	5.86	6.33	6.38
D(8 4 256 1 4 4)	(6.1 7.4)	9.94	5.84	$\boxed{4.42}$	6.13	5.42	6.35	6.16	5.86	6.31	6.38
F(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	$\boxed{6.13}$	5.72	6.35	6.16	5.86	6.33	6.38
G(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	6.13	$\boxed{5.72}$	6.35	6.16	5.86	6.33	6.38
H(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	$\boxed{9.45}$	8.39	8.93	10.09	9.61
GF(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	$\boxed{8.39}$	8.93	10.09	9.61
GEF(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	$\boxed{8.93}$	10.09	9.61
DH(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	8.93	$\boxed{10.09}$	9.61
DHEF(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	8.93	10.09	$\boxed{9.61}$

Cost=10.0 Range=10%											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
A(8 2 256 1 4 4)	(6.1 6.6)	$\boxed{9.38}$	6.15	4.33	6.13	5.72	6.35	6.16	5.86	6.33	6.38
C(8 2 256 1 4 4)	(6.1 6.6)	9.38	$\boxed{6.15}$	4.33	6.13	5.72	6.35	6.16	5.86	6.33	6.38
D(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	$\boxed{4.33}$	6.13	5.72	6.35	6.16	5.86	6.33	6.38
F(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	$\boxed{6.13}$	5.72	6.35	6.16	5.86	6.33	6.38
G(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	6.13	$\boxed{5.72}$	6.35	6.16	5.86	6.33	6.38
H(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	$\boxed{9.45}$	8.39	8.93	10.09	9.61
GF(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	$\boxed{8.39}$	8.93	10.09	9.61
GEF(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	$\boxed{8.93}$	10.09	9.61
DH(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	8.93	$\boxed{10.09}$	9.61
DHEF(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	8.93	10.09	$\boxed{9.61}$

Cost=10.0 Range=50%											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
A(8 2 256 1 4 4)	(6.1 6.6)	$\boxed{9.38}$	6.15	4.33	6.13	5.72	6.35	6.16	5.86	6.33	6.38
C(8 2 256 1 4 4)	(6.1 6.6)	9.38	$\boxed{6.15}$	4.33	6.13	5.72	6.35	6.16	5.86	6.33	6.38
D(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	$\boxed{4.33}$	6.13	5.72	6.35	6.16	5.86	6.33	6.38
F(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	$\boxed{6.13}$	5.72	6.35	6.16	5.86	6.33	6.38
G(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	6.13	$\boxed{5.72}$	6.35	6.16	5.86	6.33	6.38
H(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	6.13	5.72	$\boxed{6.35}$	6.16	5.86	6.33	6.38
GF(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	6.13	5.72	6.35	$\boxed{6.16}$	5.86	6.33	6.38
GEF(8 2 256 1 8 4)	(5.8 6.6)	6.82	6.15	4.33	4.64	5.72	6.35	6.14	$\boxed{5.97}$	6.31	6.36
DH(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	8.93	$\boxed{10.09}$	9.61
DHEF(16 4 128 1 4 8)	(3.9 8.7)	0.89	3.54	3.83	5.14	5.41	9.45	8.39	8.93	10.09	$\boxed{9.61}$

Cost=10.0 Range=∞(=60%)											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
all(8 2 256 1 4 4)	(6.1 6.6)	$\boxed{9.38}$	$\boxed{6.15}$	$\boxed{4.33}$	$\boxed{6.13}$	$\boxed{5.72}$	$\boxed{6.35}$	$\boxed{6.16}$	$\boxed{5.86}$	$\boxed{6.33}$	$\boxed{6.38}$

Table 9. Speedup results for medium cost (< 10.0) architectures.

Conference on Architectural Support for Programming Languages and Operating Systems, pages 180–192, Palo Alto, California, October 5–8, 1987. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 15(5), October 1987; *Operating Systems Review,* 21(4), October 1987; *SIGPLAN Notices,* 22(10), October 1987.

[6] T. M. Conte and W. Mangione-Smith. Determining cost-effective multiple issue processor designs. In *Proc. 1993 Int'l. Conf. on Computer Design*, pages 94–101, Cambridge, MA, October 1993.

[7] T. M. Conte, K. N. P. Menezes, and S. W. Sathaye. A technique to determine power-efficient, high-performance superscalar processors. In *Proc. 28th Hawaii Int'l. Conf. on System Sciences*, volume 1, pages 324–333, Maui, HI, January 1995.

[8] A. De Gloria and P. Faraboschi. An evaluation system for application specific architectures. In *Proceedings of the 23th Annual International Workshop on Microarchitecture and Microprogramming*, pages 80–89, Orlando, Florida, November 27–29 1990. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[9] D. Epstein. Chromatic raises the multimedia bar. *Microprocessor Report*, 9(14):23–28, October 1995.

[10] D. Epstein. IBM extends DSP performance with Mfast. *Microprocessor Report*, 9(16):1,6–8, December 1995.

[11] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150,

Cost=15.0 Range=0%											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
A(16 4 256 2 4 8)	(6.8 13.0)	$\boxed{13.06}$	5.88	3.52	5.63	4.95	9.68	8.13	8.65	9.60	9.14
C(16 4 512 1 4 8)	(7.2 14.1)	11.04	$\boxed{7.46}$	3.86	5.25	5.41	10.50	8.39	8.93	10.61	9.88
D(8 4 512 1 4 4)	(6.1 11.0)	10.72	6.07	$\boxed{4.42}$	6.13	5.42	6.35	6.16	5.86	6.31	6.38
F(8 4 512 1 4 4)	(6.1 11.0)	10.72	6.07	4.42	$\boxed{6.13}$	5.42	6.35	6.16	5.86	6.31	6.38
G(8 2 256 1 4 4)	(6.1 6.6)	9.38	6.15	4.33	6.13	$\boxed{5.72}$	6.35	6.16	5.86	6.33	6.38
H(16 4 512 1 8 8)	(6.2 14.1)	5.95	7.46	3.86	3.98	5.41	$\boxed{10.52}$	5.75	6.79	10.58	9.74
GF(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	10.50	$\boxed{8.39}$	8.93	10.61	9.88
GEF(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	10.50	8.39	$\boxed{8.93}$	10.61	9.88
DH(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	$\boxed{10.61}$	9.88
DHEF(16 8 256 1 4 8)	(7.1 12.2)	10.54	6.43	3.86	5.25	5.41	10.50	8.39	8.93	10.55	$\boxed{10.06}$

Cost=15.0 Range=10%											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
A(16 4 256 2 4 8)	(6.8 13.0)	$\boxed{13.06}$	5.88	3.52	5.63	4.95	9.68	8.13	8.65	9.60	9.14
C(16 4 512 1 4 8)	(7.2 14.1)	11.04	$\boxed{7.46}$	3.86	5.25	5.41	10.50	8.39	8.93	10.61	9.88
D(8 4 512 1 4 4)	(6.1 11.0)	10.72	6.07	$\boxed{4.42}$	6.13	5.42	6.35	6.16	5.86	6.31	6.38
F(16 4 256 2 4 8)	(6.8 13.0)	13.06	5.88	3.52	$\boxed{5.63}$	4.95	9.68	8.13	8.65	9.60	9.14
G(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	$\boxed{5.41}$	10.50	8.39	8.93	10.61	9.88
H(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	$\boxed{10.50}$	8.39	8.93	10.61	9.88
GF(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	10.50	$\boxed{8.39}$	8.93	10.61	9.88
GEF(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	10.50	8.39	$\boxed{8.93}$	10.61	9.88
DH(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	$\boxed{10.61}$	9.88
DHEF(16 4 512 1 4 8)	(7.2 14.1)	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	10.61	$\boxed{9.88}$

Cost=15.0 Range=∞(=19%)											
Arch Desc	(su,c)	A.c	C.c	D.c	F.c	G.c	H.c	GF.c	GEF.c	DH.c	DHEF.c
all(16 4 512 1 4 8)	(7.2 14.1)	$\boxed{11.04}$	$\boxed{7.46}$	$\boxed{3.86}$	$\boxed{5.25}$	$\boxed{5.41}$	$\boxed{10.50}$	$\boxed{8.39}$	$\boxed{8.93}$	$\boxed{10.61}$	$\boxed{9.88}$

Table 10. Speedup results for high cost (< 15.0) architectures.

Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[12] M. Flynn and R. Winner. ASIC microprocessors. In *Proceedings of the 22th Annual International Workshop on Microarchitecture and Microprogramming,* pages 237–243, Dublin, Ireland, September 1989. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 20(3), September 1989.

[13] B. Holmer and A. Despain. Viewing instruction set design as an optimization problem. In *Proceedings of the 24th Annual International Workshop on Microarchitecture and Microprogramming,* pages 153–162, Albuquerque, New Mexico, November 18–21 1991. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[14] I.-J. Huang and A. M. Despain. High level synthesis of pipelined instruction set processors and back-end compilers. In *Proceedings of the 29th ACM/IEEE Design Automation Conference,* pages 135–140. ACM Press, June 1992.

[15] A. Kalavade and E. A. Lee. A hardware/software codesign methodology for DSP applications. *IEEE Design and Test,* 10(3):16–28, September 1993.

[16] M. Langevin, J. Wilberg, P. Plöger, and H.-T. Vierhaus. A codesign methodology for high performance embedded systems. In V. Van Dongen, editor, *Proceedings of the High Performance Computing Symposium '95, Canada's Ninth Annual International High Performance Computing Conference and Exhibition,* pages 353–364, Montréal, Québec, July 10–12, 1995. Centre de Recherche Informatique de Montréal.

[17] J. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnel, and J. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing,* 7(1/2):51–142, May 1993.

[18] G. D. Micheli. Computer-aided hardware-software codesign. *IEEE Micro,* 14(4):10–16, August 1994.

[19] J. M. Mulder, R. J. Portier, A. Srivastava, and R. in 't Velt. An architecture framework for application-specific and scalable architectures. In *Proceedings of the 16th Annual International Symposium on Computer Architecture,* pages 362–369, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 17(3), June 1989.

[20] S. Note, W. Geurts, F. Catthoor, and H. D. Man. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *Proceedings of the 28th ACM/IEEE Design Automation Conference,* pages 597–602. ACM Press, June 1991.

[21] M. Slater. MicroUnity lifts veil on media processor. *Microprocessor Report,* 9(14):11–18, October 1995.

[22] G. Wallace. The JPEG still picture compression standard. *Commun. ACM,* 34(4):30–44, Apr. 1991.