On the Limits of Automatic Processor Specialisation by Mapping Dataflow Sections on Ad-hoc Functional Units

Paolo Ienne, Laura Pozzi, and Miljan Vuletić
Swiss Federal Institute of Technology Lausanne
Processor Architecture Laboratory
IN-F Ecublens, 1015 Lausanne, Switzerland
{Paolo.Ienne, Laura.Pozzi, Miljan.Vuletic}@epfl.ch

Technical Report CS 01/376
December 2001 — Updated June 2002

Abstract

Many available systems-on-chip embedded processors can be specialised for a given application-domain by adding ad-hoc functional units. These functional units can be used to map clusters of elementary arithmetic or logic operations (sections of the dataflow graph); more complex hardware add-ons could attack the control flow (e.g., map loops onto sequential functional units). For automatic processor specialisation, the two strategies imply different methods and have a different complexity. Before incurring in the complexities of control flow, this paper attacks the question that appears so far unanswered: how much can one improve the performance of an embedded processor on specific algorithms by automatically mapping only dataflow sections of code on special functional units? The basic scope for performance improvement is assessed and broken out in different sources: hardware parallelism; avoided quantisation of instructions in an integral number of cycles; and simplification of the logic due to constants. The scope for speedup is increased through additional manual optimisations (amenable to automation): bit-width analysis and arithmetic optimisation. Finally, ILP techniques such as loop unrolling and predication are used to increase the size of the basic blocks and give more scope for the sources of improvement. The results show that significant improvements in speed (up to 6 times) can be targeted without necessarily mapping the control flow onto hardware.

1 Introduction

The proliferation of high-performance embedded processors in *Application Specific Integrated Circuits* (ASICs) opens new opportunities for processor architecture: general purpose processors can be specialised for a given application domain by adding arbitrary *Ad-hoc Functional Units* (AFUs). Many embedded-processor architectures introduced in the recent years are to different extents manually customisable (e.g., ARC Core [8], Tensilica Xtensa [17, 14], STMicroelectronics LX [7], Infineon Carmel 20xx [6]). Essentially, processor specialisation to an application domain emerges as a compromise between fast and inflexible custom hardware and relatively slow but fully versatile software on an embedded processor (Figure 1).

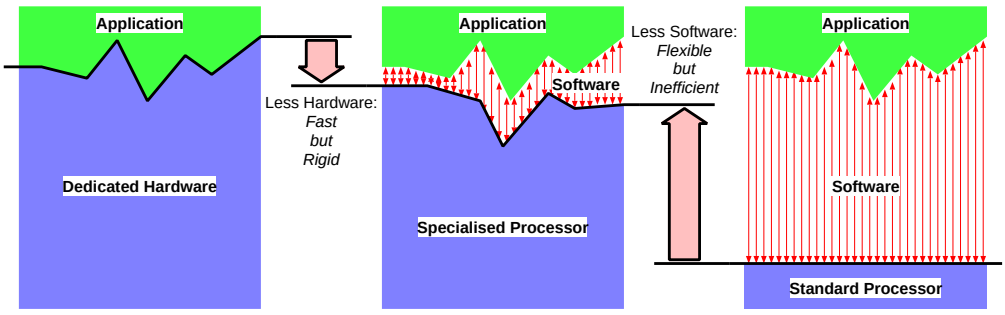

Figure 1: Specialising a processor might achieve the optimal compromise between speed and power design goals and flexibility.

For successful specialisation in the ASIC world, an automatic specialisation flow from high-level application code is needed [4]. Automation pivots around the identification of the best code sections to map on hardware. Typically, identification work has been published with a specific architecture in mind and in most cases it has been directed to a special case of AFUs: those which are implemented in a lattice configurable after manufacturing, such as FPGAs [15, 9, 1, 12, 19, 11, 2].

1.1 Goals of the Present Work

When designing compilers to generate AFUs a fundamental choice needs to be made between mapping exclusively sections of the dataflow computation (clusters of elementary arithmetic or logic operations) or mapping more complex hardware add-ons that include the control flow (e.g., complete loops onto sequential functional units). The main question which must be answered to attack the problem of automatic identification is: is there enough potential gain to map dataflow sections of user code on AFUs? Or is it a must to turn toward control flow and more complex AFUs?

This work empirically studies the fundamental limits of dataflow-based specialisation with minimal assumptions on the basic processor architecture which should be extended with AFUs. Three sets of questions should be answered by analysing typical multimedia code:

- What is the potential speedup which can be targeted by mapping dataflow-only code on ASIC AFUs?
- Can classic compiler techniques for *Instruction Level Parallelism* (ILP) or specific hardware-related optimisations increase significantly the possible speedup? How much?
- What are the key issues that need to be addressed to realise processors exploiting such specialisation possibilities?

1.2 Related Work

Several research groups have made contributions in the field of measuring benefits of specialised functional units. An influential related contribution is the CHIMAERA system [19] which reports an average speedup of 21%. Yet, at least two elements make the present work tangibly different: (1) The CHIMAERA system makes some essential microarchitectural assumptions such as the fact that AFUs have a single output and

the absence of a memory interface from the AFUs; here we want exactly to investigate the usefulness of relaxing these constraints. (2) CHIMAERA maps the AFUs on reconfigurable hardware, adding a significant technology penalty; in this work, we assume an implementation in the same ASIC technology as the main processor complex. In this respect, the present work strives to make the technology assumptions as transparent as possible.

Other data concerning specialised functional units have been published in connection with reconfigurable computing (see, for instance, [15, 9, 1, 12, 11, 2]). Again, we see the need for more general data not influenced by the specific techniques developed by the authors to identify the clusters of operations to be mapped on hardware, nor dependent on some strong microarchitectural limitations dictated by the specific hardware considered. Some of the authors [9] also explore the advantage of mapping control flow on hardware. Still, it is not clear what the maximum advantage is that can be achieved at the dataflow level. That is the topic, among others, this work attempts to investigate.

It should be noted that this paper does not address directly neither (1) the definition of the algorithms for the automatic identification of dataflow clusters nor (2) the automatic generation of the AFUs. The second is a well-understood design problem (see for instance [5] for the general case or [3] in the area of reconfigurable computing). On the other hand, the first problem has been often addressed in literature—see the references of the previous paragraph—although never fully solved.

1.3 Structure of the paper

In the next section, we illustrate the flow that we use to analyse a series of multimedia benchmarks and we describe our models for hardware and software execution. In the following section, we discuss the metrics used for benchmarks. We present and discuss our results in Section 4, and draw our conclusions in the final section.

2 Experimental Setup

Experiments are run on almost all MediaBench programs [13]. We use the SUIF compiler infrastructure [18] as the central component of our analysis flow. The overall flow is shown in Figure 2.

The original C code is compiled to the SUIF intermediate representation and annotated with profiling results. Dataflow graphs are constructed for each basic block and characterised by the metrics discussed in Section 3, using hardware and software SUIF instruction models (described below). Final results are generated by processing the available per-basic-block information.

2.1 Processor Architecture

As the baseline, unaugmented processor, we consider a single-issue pipelined processor where each instruction (that is, each SUIF node) occupies the execute stage for a single clock cycle. Dependences between successive instructions do not need to be considered in the presence of an FU-to-FU forwarding path. The processor is assumed to have no cache, neither for instructions nor data (as it is the case for many embedded processors) or, equivalently, to have perfect hit rates (which is possibly true for tight high-frequency loops—the natural focus of this type of work); loads and stores also take one cycle. Jumps and changes in control flow are approximately accounted by adding a fixed penalty to the cycle count of each basic block.

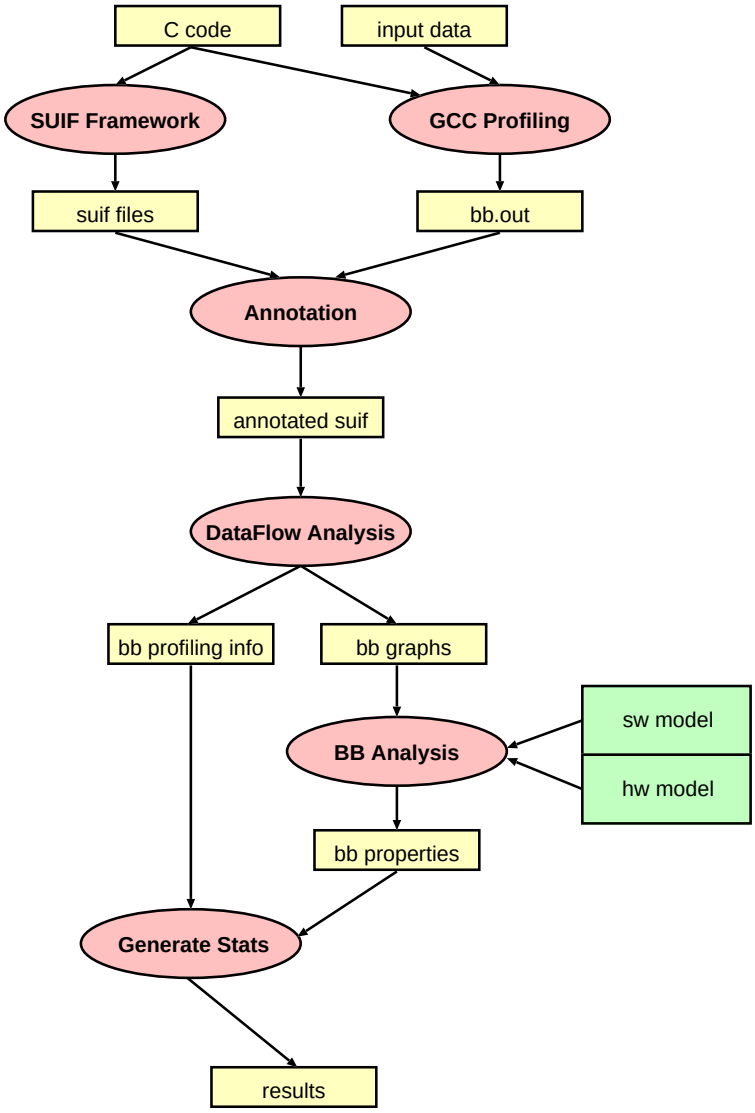

Figure 2: Analysis flow based on SUIF.

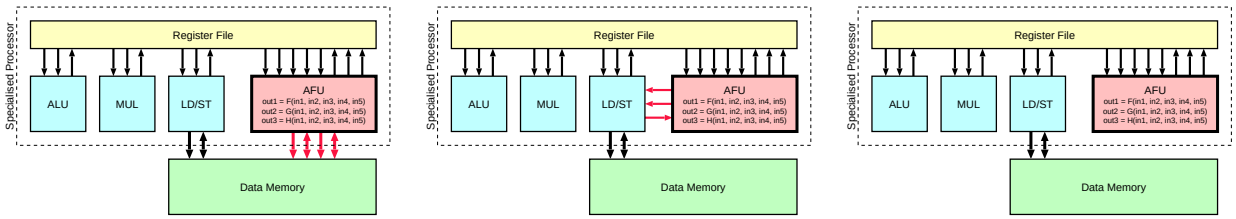

Figure 3: Processor with five-input three-output ad-hoc functional unit. (a) Direct interface with arbitrary bandwidth between Data Memory and AFU. (b) Interface via Load/Store unit and one transaction per cycle between Data Memory and AFU. (c) No interface between Data Memory and AFU.

A particularly critical architectural feature is the ability of the AFUs to access data memory. We distinguish two cases: (i) arbitrary memory bandwidth to and from the AFU, shown in Figure 3(a) and referred later as the case of *parallel memory accesses*, and (ii) single memory access per cycle to and from the AFU, shown in Figures 3(b) and 3(c), and later referred as *sequential memory accesses*.

Although approximate, we believe that this model is realistic for our estimations and it represents with acceptable accuracy a class of typical single-issue ASIC processors which could most benefit from the use of AFUs. One can view the present approach as an exploration of the chances to add automatically (and at low cost and risk) ad-hoc parallelism to inherently scalar ASIC processors.

2.2 Hardware Implementation of DFGs

In order to measure hardware speedup with reasonable accuracy, we have modelled in some details all arithmetic and logic components which could be implemented in an AFU. Latencies for every possible operation has been measured by implementation and simulation in an ASIC technology. All delays have been expressed relatively to the 32-bit multiply-accumulate delay, since a typical embedded digital signal processor is assumed with cycle time essentially determined by the multiply-accumulate operation. Table 1 shows the relative delay Λ_{hw} of some operators.

3 Metrics

The ultimate metric used to analyse the limits of dataflow-based specialisation is, of course, the speedup potential which is defined below. We also address the Instruction Level Parallelism [10] available inside a basic block because this is useful to understand to which extent parallelism in the AFUs is the source of the measured advantage.

Speedup potential. Speedup is measured by comparing the estimated cycles of transformed code partly mapped on AFU to the estimated number of cycles of the original code in software. The execution time of each basic block (in cycles) is computed (for software and hardware execution, respectively) as

$$\mathrm{T_{sw}} = \sum_{\text{all instructions}} \Lambda_{\mathrm{sw}}, \tag{1}$$

Operator	Precision	Relative Delay Λ_{hw}
Multiply-Accumulator	32 bits x 32 bits + 64 bits	1.00
Adder	4 bits + 4 bits	0.11
Adder	8 bits + 8 bits	0.12
Adder	16 bits + 16 bits	0.20
Adder	24 bits + 24 bits	0.24
Adder	32 bits + 32 bits	0.25
Divider	4 bits / 4 bits	0.38
Divider	8 bits / 8 bits	1.22
Divider	16 bits / 16 bits	3.68
Divider	24 bits / 24 bits	6.33
Divider	32 bits / 32 bits	9.61
Divider (by power of two)	any / any	0.00
Barrel shifter	8 bits	0.08
Barrel shifter	16 bits	0.11
Barrel shifter	32 bits	0.16
Barrel shifter (by constant amount)	any	0.00
Bitwise multiplexer	any	0.02

Table 1: Examples of hardware timing models of some operators. The CMOS technology used is a common 0.18μm process and the standard cells are from a popular library.

and

$$\mathrm{T_{hw}} = \lceil \mathrm{CP_{hw}} \rceil = \left\lceil \sum_{\text{instructions in the CP}} \Lambda_{\mathrm{hw}} \right\rceil . \tag{2}$$

Equation 1 expresses the fact that all instructions need to be executed sequentially on the single-issue pipelined processor (temporal computation); therefore, their latencies in the execute stage of the pipeline must be added (as discussed toward the end of Section 2.1). Equation 2 relies on the *Critical Path* (CP), and thus accounts for the fact that the AFU fully exploits the available parallelism through the appropriate hardware operators (spatial computation).

Hybrid Hardware/Software Basic Blocks. To account for the case of sequential memory accesses (see Section 2.1), we slice the basic block in successive subgraphs, or layers, composed of operations which can be executed in specialised hardware; these are interleaved by "software" layers where either the processor or the AFU itself sequentially performs memory accesses. An important metric to judge the consequent fragmentation of the basic block is the number of hardware layers after the slicing (indicated as n_{hw}).

The execution time of a basic block partly left in software (loads and stores) and partly in hardware is estimated as

$$\mathrm{T_{hyb}} = \sum_{\text{all AFU subgraph layers}} \lceil \mathrm{CP_{hw}} \rceil + \sum_{\text{all sw instr.}} \Lambda_{\mathrm{sw}}. \tag{3}$$

4 Experimental Results

In this section we review the results of the analysis on most of the MediaBench programs. We first analyse the most significant basic blocks in terms of their topological features and execution times. We then look

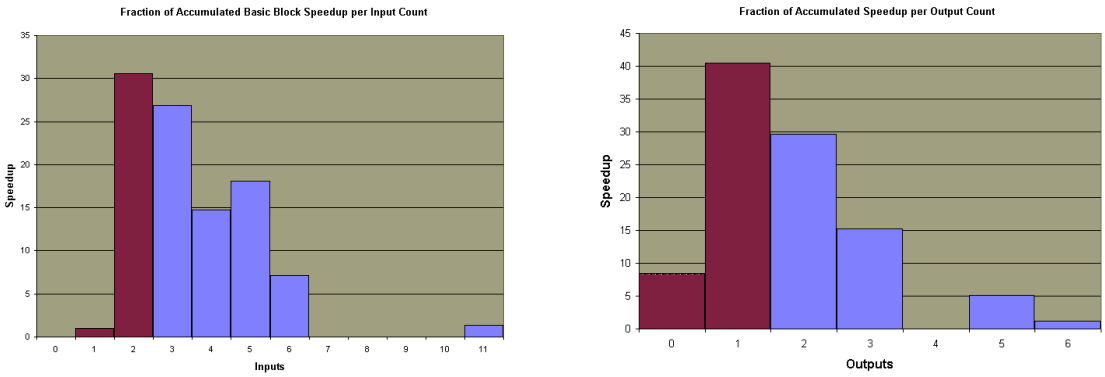

Figure 4: Histograms of the number of inputs and outputs in the basic blocks of Table 2, weighed by their speedup potential (sequential memory accesses). More than 60% of the potential speedup comes from basic blocks with more than two inputs and about 50% from basic blocks with more than a single output.

at the overall speedup potential that they convey and at the dependence of such speedup on the basic causes and on some architectural features. In a next step we apply ILP transformations to verify the possibility of enhancing the basic results and again we look into the causes of the speedup.

4.1 Characteristics of the Basic Blocks

For each benchmark considered, Table 2 shows the main characteristics of the most important basic blocks. The number of basic blocks listed is such that the sum of the time spent in them is above 60% of the total execution time. The fraction of total execution time spent in each basic block is indicated as the column "weight" in the table. One can get a first feeling for the possible improvements by comparing the software execution time $\mathrm{T_{sw}}$ to $\mathrm{CP_{hw}}$, the hardware critical path (assuming parallel memory accesses from the AFU), and to $\mathrm{T_{hyb}}$, the hardware execution time of an AFU with sequential memory access.

Moderate hardware resources required for the AFUs. The first information obtained from the table is that, in many cases, half of the execution time is spent in not more than 2–3 basic blocks. Exceptions are djpeg and gsmdecode where the execution time is more evenly distributed. Concerning the complexity, one can turn to the software execution time $\mathrm{T_{sw}}$ which approximately represents the number of instructions in the basic block: this number is typically (but not always) very small even for very important basic blocks. This indicates that the hardware resources necessary to obtain the potential speedups discussed later are small or very small, probably never unrealistic.

Pressure on the register file higher than classically supported. An important point to address is the pressure that AFUs implementing complete basic blocks would create on the register file. Some perspective on the incremental advantage of supporting more inputs or outputs comes from the histograms of Figure 4. They are obtained from the data of Table 2 and from speedup data by accumulating the potential speedup of all basic blocks with a given number of inputs or outputs. One can observe that more than 60% of the potential gain comes from basic blocks with more than two inputs and about 50% from basic blocks with more than a single output. Clearly the classic model of functional unit with 2–3 inputs and 1 output does not fit very well. Providing the AFUs with several read and write ports to the register file is a key ingredient of

Benchmark		Weight	Topology				Parallel Memory Accesses		Sequential Memory Accesses	
	#		In	Out	Ld	St	T_{sw}	CP_{hw}	n_{hw}	T_{hyb}
adpcmdecode	5	22.84%	3	2	1	0	9	2.07	2	3
adpcmdecode	22	17.77%	4	3	1	1	7	1.25	1	3
adpcmdecode	9	12.69%	2	3	0	0	5	0.33	1	1
adpcmdecode	4	7.61%	1	3	1	0	6	1.00	2	3
djpeg	10	20.96%	5	5	0	0	7	0.83	1	1
djpeg	4	9.58%	11	2	10	3	34	5.24	3	16
djpeg	8	8.98%	2	0	1	0	3	1.58	1	2
djpeg	7	6.34%	6	6	2	2	23	3.49	2	8
djpeg	9	5.52%	3	1	18	8	112	5.80	2	30
djpeg	11	3.74%	3	3	0	0	5	0.83	1	1
djpeg	94	3.03%	3	2	0	0	3	0.50	1	1
djpeg	4	2.13%	3	3	16	8	118	5.52	2	29
epic	45	47.30%	6	3	2	0	9	2.47	2	5
epic	13	9.99%	3	1	1	0	68	19.27	2	21
epic	46	8.37%	5	2	0	0	3	0.50	1	1
g721encode	3	31.23%	2	1	1	0	5	1.25	2	3
g721encode	4	16.84%	2	1	0	0	3	0.83	1	1
g721encode	11	5.16%	3	2	0	0	12	1.08	1	2
g721encode	49	3.55%	2	0	2	1	11	2.75	2	5
g721encode	56	2.96%	2	1	1	1	11	2.50	1	3
g721encode	3	2.42%	2	2	2	0	9	1.50	2	4
gsmdecode	21	11.97%	3	1	1	0	8	2.32	2	4
gsmdecode	3	11.97%	3	2	2	0	8	1.90	2	4
gsmdecode	27	8.98%	3	1	0	1	6	1.49	1	2
gsmdecode	20	7.48%	2	1	0	0	5	1.00	1	1
gsmdecode	8	7.48%	2	1	0	0	5	1.00	1	1
gsmdecode	9	7.48%	2	1	0	0	5	0.83	1	1
gsmdecode	15	5.98%	2	2	0	0	4	0.65	1	1
gsmencode	24	53.07%	3	1	80	0	360	12.69	2	93
gsmencode	3	9.32%	5	3	2	1	16	3.32	2	7
mpeg2decode	4	37.44%	5	2	2	0	13	2.49	2	4
mpeg2decode	10	34.56%	4	2	2	0	12	2.49	2	4
pegwit	1	31.47%	2	0	296	36	811	3.65	3	335
pegwit	25	9.06%	5	2	0	0	7	0.83	1	1
pegwit	28	6.47%	2	0	2	1	5	2.29	2	5
pegwit	9	6.45%	2	1	2	0	5	2.82	3	5
pegwit	13	6.45%	2	0	2	1	5	2.29	2	5
pegwit	10	5.16%	4	2	0	0	4	0.83	1	1
unepic	57	38.00%	2	1	1	0	68	19.28	2	21
unepic	4	9.43%	4	1	0	0	34	9.95	1	10
unepic	10	9.07%	4	0	9	9	50	3.00	2	20
unepic	5	4.45%	4	1	0	1	6	1.55	1	3

Table 2: Basic characteristics of untransformed basic blocks. The table shows, for each benchmark, a number of basic blocks such that their total weight (that is number of cycles spent in them) is above 60%.

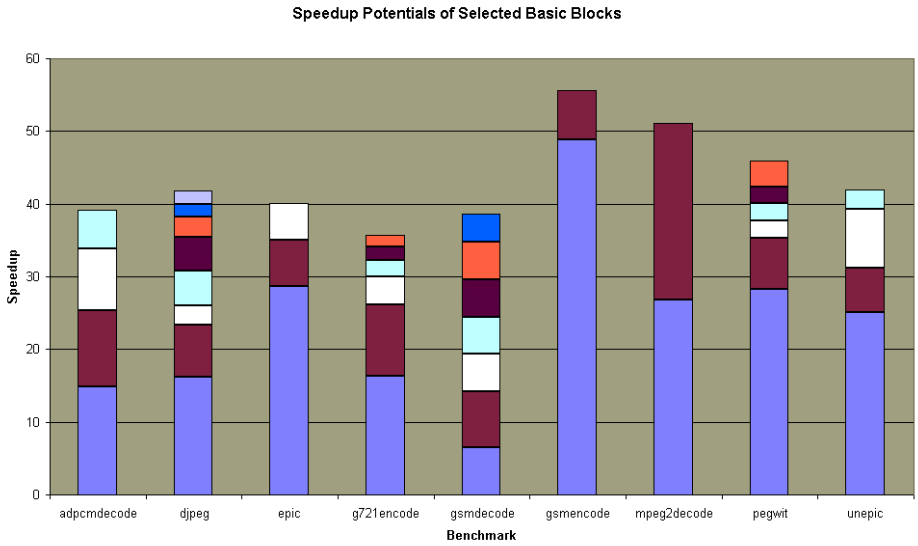

Figure 5: Speedup achievable by mapping the most important basic blocks on AFUs. Note that to achieve approximately $1.5\times$ speedup, only 2–3 AFUs are needed in most cases, and 5–6 at worst.

success. The number of inputs has been raised in previous work (e.g., nine [19]); our basic data indicate that up to six ports should be sufficient. Conversely, the number of outputs has constantly been kept to one but it could be conveniently raised.

Limited importance of memory ports in the AFUs (I). The last key indication from Table 2 concerns the importance of parallel paths from the AFU to data memory—see Figure 3(a). Two pieces of information are relevant: Firstly, with a couple of extreme exceptions, the number of memory accesses is quite low, often between none and two. Secondly, the number of hardware layers n_{hw} obtained by "carving out" memory accesses from the basic blocks is with a few exceptions either one (indicating that the loads or stores can be concentrated all at the beginning or end of the basic block) or two (which typically corresponds to a "thin" first layer for address calculation, a second layer of memory load(s), followed by a more significant subgraph processing the data). A preliminary conclusion is that sequential memory accesses should not lead to a major speedup drop.

Small delay of typical basic blocks. The delay of the basic blocks—before rounding and factoring out the layers of memory accesses—are often very small and below unity (corresponding to the already remarked simplicity of the basic blocks). This (1) suggests that the avoided quantisation of time in integral clock cycles for each operation might be an extremely significant source of gain and (2) indicates that further optimisation efforts to reduce the AFU critical paths in a VLSI implementation may not prove very useful since in most cases the rounding to the next integral cycle will hide the benefit.

4.2 Potential Speedup

Figure 5 shows the potential reduction of execution cycles achievable by mapping all basic blocks of Table 2 on AFUs. The following comments can be made.

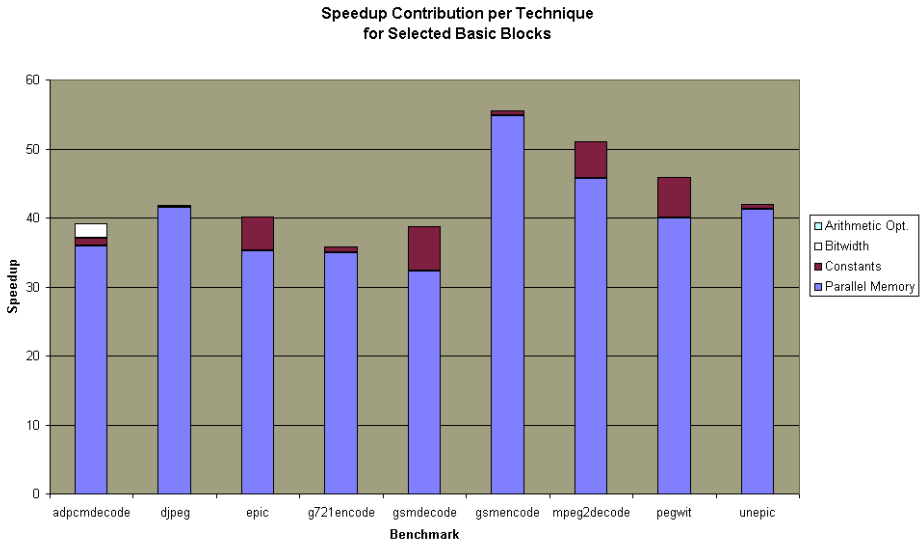

Figure 6: Contribution to the speedup of hardwiring constants. For adpcmdecode bitwidth analyses and arithmetic optimisations have also been performed and shown.

Limited available parallelism. The Instruction Level Parallelism, although not explicitly shown in the graph, was generally extremely low, hardly above two, even in presence of nonnegligible potentials to save execution cycles. In those cases where a tangible parallelism was measured, this is invariantly due to memory accesses; yet, parallelism due to several independent memory accesses is not of great interest. One of the potential sources of gain for AFUs (that is, parallelism in hardware or spatial computation) goes thus far unexploited.

Reasonable but not very large basic speedup. Basic cycle savings (parallel memory bar with hardwired constants) range between 10 and 50%; typical values are around 20–25%. The number of basic blocks that must be mapped on an AFU to achieve 30% of cycle savings (a speedup of approximately $1.5\times$) range from 2–3 in most cases to 5–6 at worst. The potentials for advantages are there but are not (yet) fully satisfactory.

Hardwired constant values not a key advantage. Hardwiring constants is a high element of design risk. On the other hand, one could believe that hardwiring constant operators is an essential ingredient of the AFU advantage since many typical operations such as bitwise logic operators, shifts, etc. reduce to no operation at all in the case of a constant input. Note that "variable" constants does not imply an increase of the basic block inputs (e.g., they can be programmed into power-up configuration registers). In fact, Figure 6 results show that avoiding constant hardwiring appears a very small price to pay for a sensible reduction in the design risk.

Limited importance of memory ports in the AFUs (II). As Figure 7 shows, the loss in speedup when sequential access is considered, compared to parallel access, is up to 15–20% and, although not negligible, it is never really dramatic. Note also that the most consistent losses (e.g., in gsmencode and pegwit) are where the parallel access cases cycle savings correspond to the quite unrealistic situation of a very large number of memory operations in parallel. Although the importance of memory ports in the AFUs cannot be

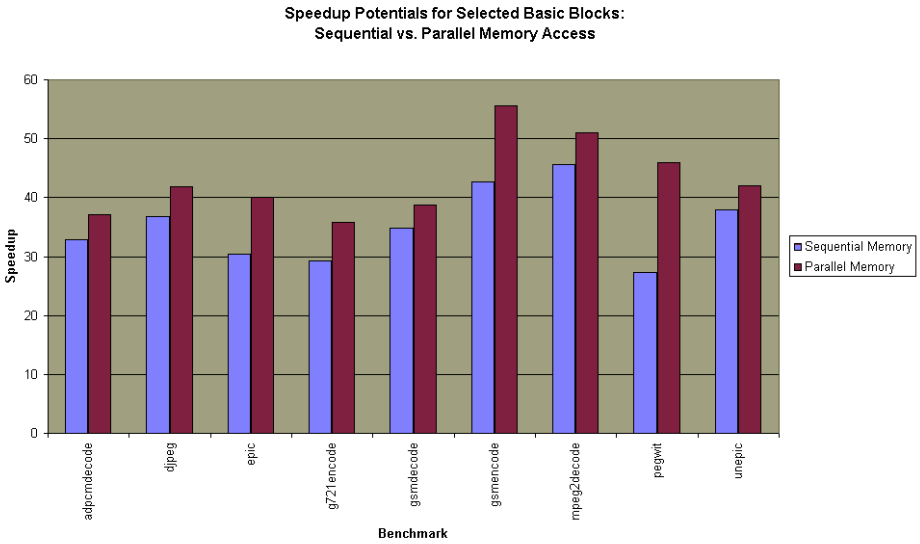

Figure 7: Comparison of total speedup with either sequential or parallel memory accesses. Although memory ports in the AFUs improve their effectiveness, the microarchitectural complexity may exceed the limited advantages.

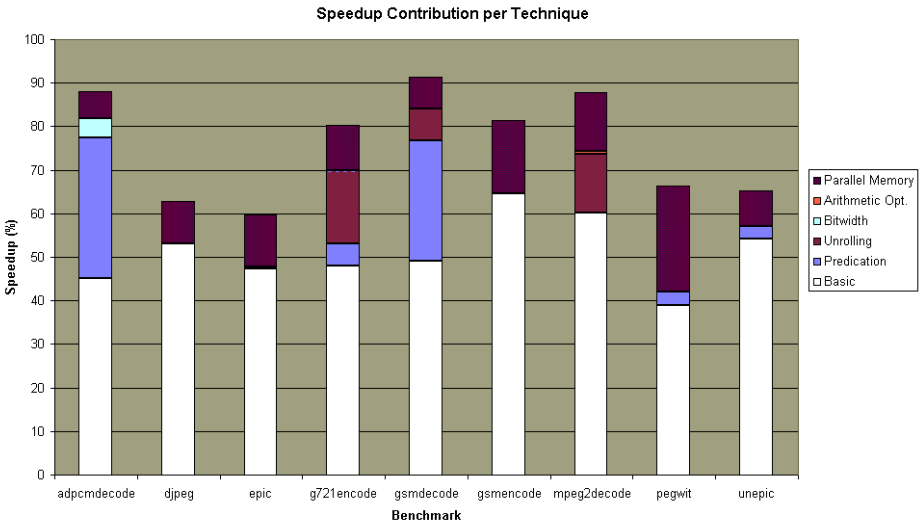

Figure 8: Speedup contribution per optimisation technique when ILP transformations are performed.

Benchmark		Cycle Savings	Topology				ILP	Parallel Memory Accesses		Sequential Memory Accesses		
	#		In	Out	Ld	St		T_{sw}	CP_{hw}	n_{hw}	T_{hyb}	AFU Area
adpcmdecode	2	77.51%	8	8	4	1	4.73	156	4.32	2	10	3.17
epic	13	7.23%	5	1	1	1	1.09	80	20.23	2	21	1.74
g721encode	1	47.08%	2	1	15	0	12.27	772	4.31	2	21	6.29
gsmdecode	2	70.93%	5	2	25	10	7.05	712	11.73	2	46	29.40
mpeg2decode	3	34.31%	3	1	16	1	5.75	69	4.98	2	21	9.76
mpeg2decode	7	25.01%	2	2	16	0	5.00	55	4.23	2	20	9.46
pegwit	25	8.30%	5	0	2	1	1.63	18	3.07	2	6	0.57
pegwit	10	6.20%	4	0	2	1	1.36	15	3.07	2	6	0.43
unepic	57	29.65%	2	1	1	1	1.15	85	20.25	2	22	2.67

Table 3: Basic characteristics of basic blocks transformed with either predication, loop unrolling, or both (as applicable). The table shows, for each benchmark, the basic blocks resulting from the transformations: they do not correspond to those of Table 2 and are not necessarily the most important ones in term of potential speedup or weight. AFU area is also measured relative to the area of a 32-bit multiply-accumulate.

denied in all cases, it does not appear as an essential feature.

Limited or no advantage of accurate bitwidth analyses and arithmetic optimisations (I). For adpcmdecode, the subgraphs of the best basic blocks have been manually annotated with the exact precision of the variables as it can be deduced from the code. Techniques have been described in literature to perform automatically the same task [16]. Although the critical path CP_{hw} does indeed reduce, the execution time decreases for only one of the considered basic blocks because of the rounding to the next integral number of cycles. The limited impact on the speedup is shown in Figure 6. The impression is that possibilities of tangible arithmetic optimisations rarely exist in these very simple basic blocks.

4.3 ILP Transformations

The results of the previous sections indicate that parallelism lays largely unexploited. The natural next step is therefore to apply some classic techniques used to expose instruction level parallelism [10]. We have chosen the simplest and, in our view, most promising ones: predication and loop unrolling.

The ILP techniques have been applied mimicking what an automatic preprocessing path could do. Specifically, predication has been performed by introducing a predication variable that holds the result of a condition evaluation. The variable is used to transform a conditional control structure into predicated assignment, where all possible outcomes of the structure are computed, but only the appropriate one is assigned. Loop unrolling has been applied only to loops with a constant and limited number of iterations (up to 8) that consist of a single basic block (possibly after predication). The modified code has been run and validated.

The cycle savings potentials due to each additional transformation or optimisation are shown in Figure 8. To visualise benefits of AFU memory interface, the advantage of having parallel memory access is shown as if it were an additional technique. Figure 9 shows the contribution of each basic block to the overall speedup (using all techniques but with sequential memory access). One can notice that the important basic blocks in terms of speedup are only the 1–3 largest ones, and the contribution of further basic blocks is small or irrelevant. In other words, by adding only 1–3 AFUs, one obtains most of the speedup available. Table 3

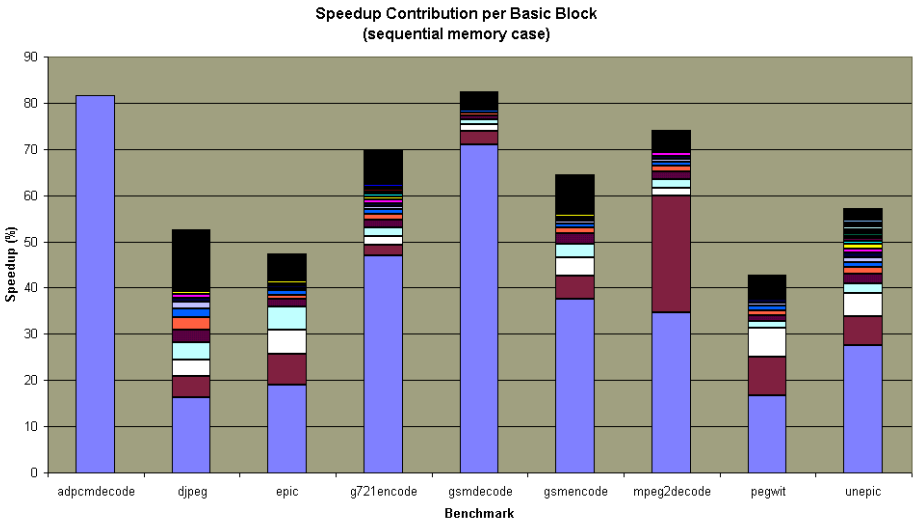

Figure 9: Speedup contribution per basic block after ILP transformations in the case of sequential memory accesses. Note that now to achieve approximately $1.5\times$ speedup, only a single AFU suffices in most cases.

shows the usual characteristics of the predicated and/or unrolled basic blocks, and their area estimation.

Major improvements through predication and unrolling. Although on a few benchmarks the ILP compilation techniques are almost or completely useless, their combination is in general rather effective on two fronts: reduce the fragmentation of important kernels in many basic blocks and therefore increase, at times significantly, the potential speedup. In all but three cases, after both predication and unrolling, the top basic block alone mapped on an AFU with sequential memory access can bring more than 30% cycle reduction. Cumulative speedups, again in the realistic case of sequential memory access, range between a minimum of $1.7\times$ up to $6.3\times$.

Figure 10 shows that the pressure on the register file is increasing with respect to Figure 4. Yet, workarounds might be possible after the ILP transformations—e.g., a sequentialisation of the register file accesses (probably less damaging now that the average $\mathrm{CP}_{\mathrm{hw}}$ of the new basic blocks has increased).

Reasonable area investment. The estimation of the area occupied by AFUs is done relative to the area of a 32-bit multiply-accumulate. The results are show in Table 3. As an example, the top basic block in adpcmdecode has 19 adders, 11 comparators, and many bitwise operators. The hardware complexity of the two top basic blocks in mpeg2decode is dominated by eight multipliers.

Limited or no advantage of accurate bitwidth analyses and arithmetic optimisations (II). Even on the much larger basic blocks now available, accurate bitwidth analyses and arithmetic transformations fail to bring a significant advantage, at least on a few cases considered, even when they effectively reduce the hardware delay of a basic block. Yet, this time the source of the missing improvement is different. Consider the main section of both dominating basic blocks of mpeg2decode, shown in Figure 11: it represents the scalar product of two eight-element vectors. The natural arithmetic transformation consists (1) in converting the chained reduction sums (resulting from loop unrolling) into a binary addition tree and (2) in performing

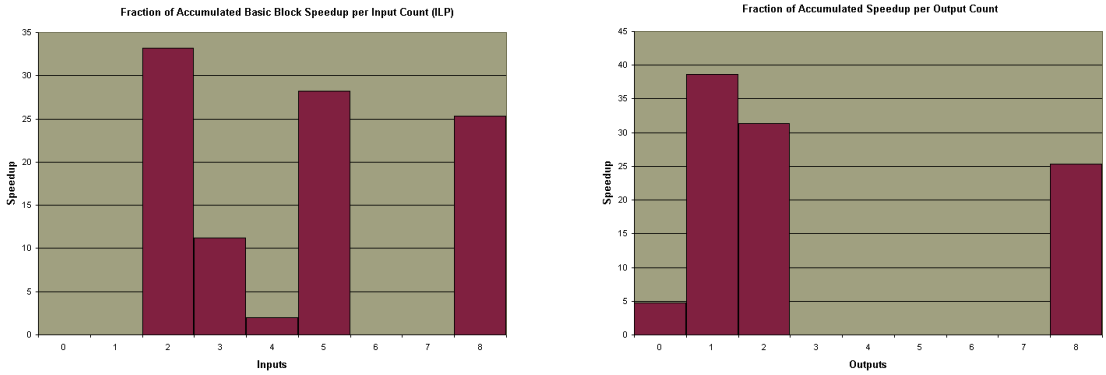

Figure 10: Histograms of the potential speedup per input and output count, after ILP transformations. Compared to Figure 4 multiple register-file writes are slightly reduced and the importance of several reads increased.

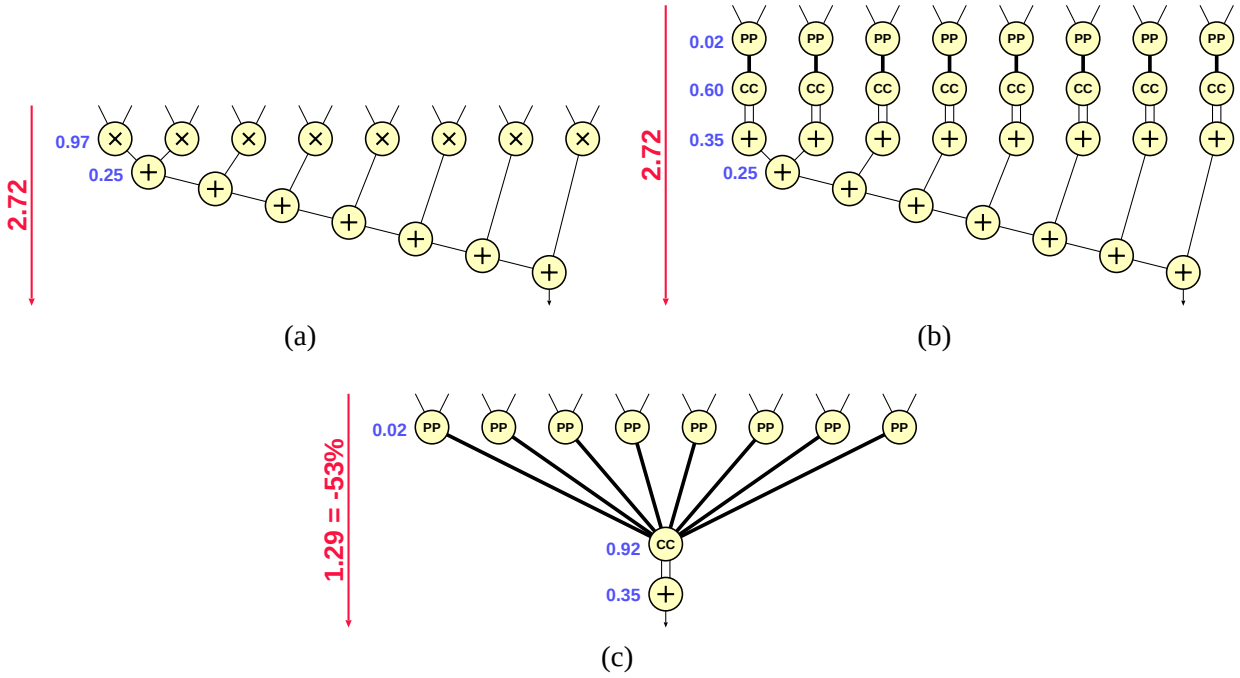

Figure 11: Slightly simplified main subgraph of both dominant mpeg2decode basic blocks before and after arithmetic transformations: (a) original subgraph; (b) actual implementation using partial product generators (PP), column compressors (CC), and carry-propagate adders; (c) optimised implementation, more than twice faster. The numbers indicate the relative delay Λ_{hw} and the critical path delay $\mathrm{CP}_{\mathrm{hw}}$.

all sums but the last one (including the final sums internal to the combinatorial multipliers) using carry-save adders. All partial products generated by the multipliers front-ends are fed to a large column compressor and finally added with a carry-propagate adder. The critical path improves tangibly (e.g., for basic block 7 with parallel memory access, from 4.23 to 2.77 timing units) with practically no additional area cost. Yet, the additional cycles saved are not many compared to those saved by the already applied techniques:

mpeg2decode basic block #7	T_{sw}	T_{hw}	η_{bb}
No Arith. Opt.	55	5	30.6%
With Arith Trans	55	3	31.4%

It is now the gain from the high parallelism achievable in hardware (ILP= 5.00) that is hiding the benefits of arithmetic optimisations. These results suggest that for ASIC technologies, bitwidth analysis is more likely to be important for area and power saving than for performance. On the other side, arithmetic transformations are not necessarily saving area, are applicable in relatively special situations, and may not see their potentials adequately represented due to other sources of speedup.

5 Conclusions

Compared to other results reported so far in processor specialisation, the present work makes two main contributions: (1) We have inverted the process and instead of designing an architecture and proving the advantages available for that machine, we explore the potentials available at the simplest conceptual level of specialisation to direct our and hopefully others' future architectural decisions. (2) We have developed and used a relatively accurate timing model of the ASIC hardware used for specialisation.

From our analyses we conclude the following:

- There is enough scope at dataflow level to get decent speedups (2–3$\times$ in most cases) at low cost (at times a few thousand gates, often much less) and low risk (the performance of the host processor is essentially unaffected). The cost in terms of infrastructure is low—e.g., the compiler support is readily available in some processors, such as Lx or Xtensa; it is available with some restrictions in others, such as Carmel.

- A larger number of write ports (at least 2–3) from the AFUs to the register file appears very important. It is possibly the highest component of cost at microarchitectural level.

- Hardcoding of constants is a minor component of gain but a serious component of risk for real applications which might need late changes.

- Memory interfaces in the AFUs can be beneficial but are certainly not essential. A detailed cost/benefit analysis might show when the additional design burden should be taken.

- Predication and unrolling are fundamental for parallelism and thus to achieve desired results.

- Detailed techniques of bit-width analysis and arithmetic optimisations carry definite and significant intrinsic advantages but these are often masked by other effects (quantisation or parallelism). Yet, the intrinsic advantages may become visible and essential in AFU implementation technologies other than ASIC (e.g., FPGAs).

The effects of AFUs on power consumption have been disregarded in the present research. These effects may be extremely interesting: there are reasons to believe that important savings could be achieved by avoiding most register reads and writes, by reducing the accesses to instruction memory due to code compaction, etc. We plan this as our next analysis step.

References

[1] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG based design approach for reconfigurable VLIW processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 778–79, Mar. 1999.

[2] M. Arnold and H. Corporaal. Designing domain specific processors. In *Proceedings of the 9th International Workshop on Hardware/Software Codesign*, pages 61–66, Copenhagen, Apr. 2001.

[3] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 70–80, Napa Valley, Calif., Apr. 1999.

[4] A. Cataldo. Compiler that converts C-code to processor gates advances. *EE Times*, 23 Oct. 2001. http://www.eetimes.com/story/OEG20011023S0028.

[5] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[6] J. Eyre and J. Bier. Infineon targets 3G with Carmel2000. *Microprocessor Report*, 17 July 2000.

[7] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 203–13, Vancouver, June 2000.

[8] T. R. Halfhill. ARC Cores encourages “plug-ins”. *Microprocessor Report*, 19 June 2000.

[9] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Napa Valley, Calif., Apr. 1997.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, Calif., second edition, 1995.

[11] M. F. Jacome, G. de Veciana, and V. Lapinskii. Exploring performance tradeoffs for clustered VLIW ASIPs. In *Proceedings of the International Conference on Computer Aided Design*, pages 504–10, San Jose, Calif., Nov. 2000.

[12] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 1999.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–35, Research Triangle Park, N.C., Dec. 1997.

[14] S. Leibson. Vector DSP, FPU extend Xtensa. *Microprocessor Report*, 19 June 2000.

[15] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 172–80, San Jose, Calif., Nov. 1994.

[16] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 108–20, Vancouver, June 2000.

[17] J. Turley. Tensilica CPU bends to designers' will. *Microprocessor Report*, 8 Mar. 1999.

[18] R. Wilson, C. French, C. Wilson, J. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, M. Tseng, Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29:31–37, Dec. 1994.

[19] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–35, Vancouver, June 2000.