The Effect of FPGA Size on Software Speedup from Hardware/Software Partitioning

Shawn Nematbakhsh, Greg Stitt, and Frank Vahid[*]

Department of Computer Science and Engineering

University of California, Riverside

{snematbakhsh | gstitt | vahid}@cs.ucr.edu, http://www.cs.ucr.edu/~vahid

[*]Also with the Center for Embedded Computer Systems at UC Irvine

Contact Information:

Shawn Nematbakhsh, phone: 909-787-2373 fax: 909-787-4643 e-mail: snematbakhsh@cs.ucr.edu

Greg Stitt, phone: 909-787-2373 fax: 909-787-4643 e-mail: gstitt@cs.ucr.edu

Frank Vahid, phone: 909-787-4710 fax: 909-787-4643 e-mail: vahid@cs.ucr.edu

The Effect of FPGA Size on Software Speedup from Hardware/Software Partitioning

Shawn Nematbakhsh, Greg Stitt, and Frank Vahid[*]
Department of Computer Science and Engineering
University of California, Riverside
{snematbakhsh | gstitt I vahid}@cs.ucr.edu, http://www.cs.ucr.edu/~vahid
[*]Also with the Center for Embedded Computer Systems at UC Irvine

Abstract

We examine the relationship between FPGA size and software speedup when an on-chip FPGA is used to implement critical software loops through hardware/software partitioning. We studied seven benchmark programs taken from Mediabench and Netbench. We profiled the programs on the SimpleScalar architecture, rewrote the critical loops in VHDL, synthesized and mapped those loops to a Xilinx FPGA, and calculated the gate requirements and performance speedups. We created several versions of each program, each version having successively more critical code moved to the FPGA, to see the relationship between size and speedup. Our results show that surprisingly few FPGA gates are needed to obtain most of the reasonably achievable speedup – an average speedup of 6x was obtained with only about 20,000 gates.

Keywords

Hardware/software partitioning, system-on-a-chip, platforms, configurable logic, FPGA, software speedup, codesign.

1. Introduction

Single chip platforms incorporating a microprocessor and FPGA are growing in popularity. Several such platforms have become available commercially, including Atmel's FPSLIC family [2], Triscend's E5 and A7 [13], Xilinx's Virtex II Pro [16], and Altera's Excalibur line [1].

Moving frequently executed pieces of code from software to an FPGA can increase the speed of the system [4][9][10][15]. These speedups can even translate to significant energy savings [5][12].

Such speedups are typically due to the fact that a large set of software instructions, requiring perhaps tens of hundreds of clocks cycles, can often be executed in custom hardware using just a few clock cycles. The reduction in cycles comes primarily from executing instructions in parallel and from loop unrolling. Some cases allow for more speedup than others. For example, a function that does many bit operations dependant on one another could be sped up greatly. A function that has to access memory every other cycle has less potential to be sped up.

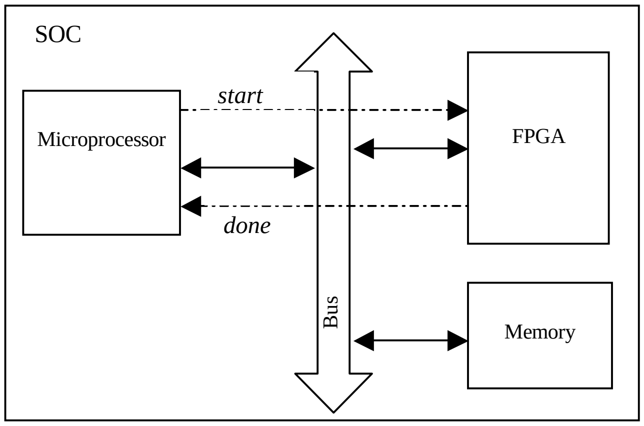
In many programs, a few small sections of code may account for a very large portion of the total execution time. This situation is advantageous for partitioning, since we need only speed up those few critical sections to gain most of the reasonably available speedup. Most of the program can remain in software - moving everything to gates would require an FPGA with unreasonable size and power requirements for most applications.

This paper examines the relationship between FPGA size and software speedup for several benchmark software applications selected from the Mediabench [6] and Netbench [8] benchmark suites. We show that relatively few gates are required to obtain most of the reasonably achievable speedup.

2. Hardware/Software Partitioning Method

We considered a straightforward approach to hardware/software partitioning, in which critical software loops are moved to an on-chip FPGA. Such an approach can be readily automated, and in fact several prototype and commercial tools, like Synopys' Nimble Compiler attempt [7] and Proceler [11], perform such partitioning. We point out that approaches that actually rewrite the algorithm for hardware execution will result in much greater speedups while requiring much more hardware. Although we actually consider both critical loop and critical subroutines, we use the term "loops" throughout the paper for simplicity.

Figure 1: System-on-a-chip architecture.



Our target system-on-a-chip architecture is shown in Figure 1, based on the Triscend E5 architecture. When a partitioned section of code is encountered, the microprocessor enables the FPGA. The FPGA then fetches the needed data from memory across the 32-bit bus, performs a computation, writes back to memory once completed, and indicates completion to the processor.

We compiled each benchmark to a SimpleScalar [3] binary. We then ran SimpleScalar's *sim-cache* to generate a trace file, and *objdump* to convert the binary to human-readable assembly. We recorded the number of cycles in software as *cycles_s*.

We created multiple hardware/software versions of some benchmarks, each version successively requiring more hardware gates. For each version, we computed cycles as *cycles_hs*. We computed speedup as *cycles_s* / *cycles_hs*. We assumed the microprocessor and FPGA used the same clock frequency, as is the case in Triscend's E5 and A7 devices [13]. We assumed the microprocessor had a cycles per instruction (CPI) of 1.5, which we observed to be a typical CPI when running a microprocessor simulator.

The hardware/software versions were created as follows: First, to detect critical loops, we used a tool called *LOOAN* (Loop Analysis) [14], which takes the output of *objdump* and the trace file, and determines the portions of code that consume the most CPU time. To map loops to hardware, we rewrote the critical sections of code in VHDL, synthesized, and then mapped to a Xilinx xcv100e FPGA using Xilinx's ISE Webpack [16]. This tool directly reported the number of gates and cycle information for a design.

For memory accesses, we assumed that the FPGA could access any location in memory in the same time required by the microprocessor - as is the case with Triscend's devices. Since the memory locations of variables were pre-known in the benchmarks, we hard coded those addresses into the FPGA.

In converting the commonly executed loops to VHDL, we performed loop unrolling wherever possible, subject to the limitation that the FPGA could be clocked at a minimum of 40 MHz - so in some cases we could not unroll completely.

3. FPGA Size and Speedup

3.1 Experiments

Table 1 summarizes the benchmarks and our hardware/software partitioning results. The *Cycles* column displays the total number of software cycles the benchmarks required without any partitioning. *#* is the number we've assigned to the loop, for reference later. *Critical Section* lists the function in which the critical loop was found. *Lines* reports the number of lines in C code of the critical loop. *Execution time* lists the percentage of execution time in software that was spent in the critical loop. *Cumulative Speedup* shows the speedup we observed after moving the critical loops to hardware. *Ideal Speedup* reports the best possible case for speedup, obtained if all critical loops in hardware were reduced to zero time. *Cumulative Gates* displays the number of gates used by the FPGA to implement the critical loops in hardware. Due to time constraints, we did not always move multiple loops to hardware. Only if the execution time of subsequent loops looked promising for speedup did we implemented them. We'll now summarize each benchmark briefly.

G721 is an audio format that is used for encoding voice. Here we found one loop that consumed 44.5% of execution and another that consumed 10.1% of execution. The first loop, inside the *quan* function, was a short *for* loop that searched through an array. The second loop, *update*, was a longer *for* loop that did some bit operations.

ADPCM, an algorithm used for speech compression, was a unique case because we discovered that a single loop was consuming 99.9% of execution time. We sped this loop up, allowing us to obtain a speedup of 27 with only 14,000 gates.

Pegwit is a program used for public key encryption. Here we found two loops that each consumed about 35% of execution time. In addition, we found two other loops consumed about 4% and 3% of execution time respectively. 4% and 3% do not look like promising loops to move to hardware, but once the two 35% loops were moved to hardware, the two smaller loops represented a significant portion of remaining execution time.

DH is another public key encryption application. This benchmark has three similar functions consuming the most execution time. The *NN_DigitMult* function performed bit multiplication on 32-bit integers. The *NN_SubDigitMult* function performed bit subtraction, and the *NN_AddDigitMult* did shifts and adds on several integers. These functions were easy to speed up in hardware.

Table 1: Speedup and FPGA size for critical loops.

Benchmark	Cycles	#	Critical Section	Lines	Execution %	Cum. Speedup	Ideal Speedup	Cum. Gates
G721	838,230,001	1	quan	3	44.5%	1.8	1.8	1,307
	838,230,001	2	update	12	10.1%	2.2	2.2	5,811
	838,230,001	3	predictor_zero	2	4.3%	N/A	N/A	N/A
ADPCM	32,894,094	1	coder	43	99.9%	27.2	1175.5	14,132
	32,894,094	2	main	10	3.6E-04	N/A	N/A	N/A
	32,894,094	3	read	N/A	9.3E-05	N/A	N/A	N/A
Pegwit	42,752,919	1	gfAddMul	5	35.4%	1.5	1.6	4,301
	42,752,919	2	gfMultiply	6	35.4%	3.0	3.4	13,419
	42,752,919	3	gfReduce	5	4.2%	3.4	4.0	15,678
	42,752,919	4	gfAdd	3	2.8%	3.7	4.5	18,150
DH	1,793,032,156	1	NN_DigitMult	16	40.4%	1.6	1.7	15,308
	1,793,032,156	2	NN_SubDigitMult	10	17.9%	2.3	2.4	20,792
	1,793,032,156	3	NN_AddDigitMult	10	16.9%	3.7	4.0	21,383
MD5	5,374,033	1	MD5_memset	2	13.4%	1.1	1.2	2,036
	5,374,033	2	Decode	3	11.0%	1.3	1.3	2,228
	5,374,033	3	MD5_Transform	71	32.3%	2.1	2.3	90,074
TL	57,412,470	1	rn_addmask	2	50.9%	1.9	2.0	5,478
	57,412,470	2	rn_search	7	4.4%	N/A	N/A	N/A
	57,412,470	3	_wordcopy_fwd_alligned	N/A	4.0%	N/A	N/A	N/A
URL	27,353,017	1	calculate_bm_table	2	80.0%	4.3	5.0	2,929
	27,353,017	2	calculate_bm_table	3	4.0%	N/A	N/A	N/A
	27,353,017	3	find_lcs	5	3.8%	N/A	N/A	N/A

MD5 is a checksum algorithm used on network packets. The most critical section of the *MD5* benchmark was the *MD5_transform* function. This was a very long function, however, and took 90,000 gates to implement in hardware. For this reason, we present the partition with that function in hardware last in the graphs and tables, after two other loops that have a better speedup to gate ratio.

In *TL*, a benchmark that does table lookups, there was a single *for* loop that consumed 50.9% of execution. This loop took eight cycles per iteration in software, and was reduced to only one in hardware.

URL is a program that does *URL* packet switching. This benchmark had a single loop that consumed 80% of execution time. This *for* loop took 16 cycles per iteration in software, but was reduced to a single cycle in hardware.

Figure 2 shows the speedup of all seven benchmarks as critical sections are moved to the FPGA. The horizontal axis shows the number of FPGA gates required by the critical loops in hardware, while the vertical axis shows the cumulative speedup.

Figure 2: The relationship between FPGA size and speedup for the examined benchmarks, obtained through hardware/software partitioning.

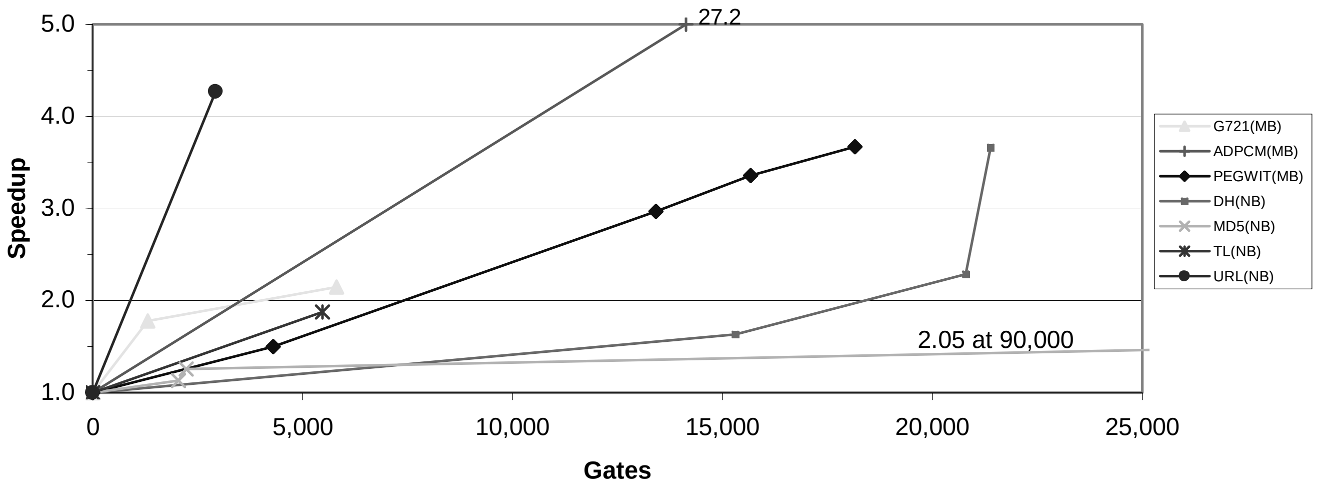


Figure 3: Average speedup for different FPGA sizes.

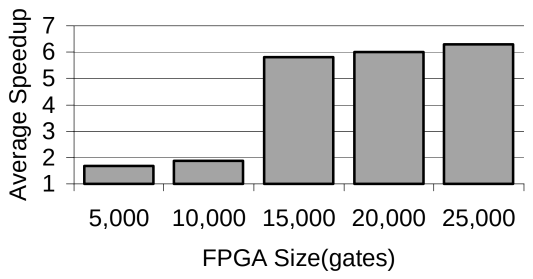
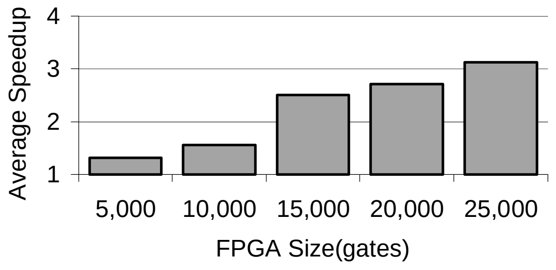


Figure 4: Average speedup for different FPGA sizes, excluding the best and worst of our benchmarks.



3.2 Discussion

The first observation we might make is that good speedup can be obtained with a relatively small amount of FPGA. Figure 3 shows the average speedups obtained for a given size of FPGA. We see that with only 20,000 gates, we obtain an average speedup of 6.0; 25,000 gates yields a speedup of 6.3. Furthermore, we see in Figure 2 that for the *adpcm* benchmark, a very good speedup of 27.2 is obtained, with less than 15,000 gates.

Because the speedup for *adpcm* is much higher than the other examples, we show the average speedups versus FPGA size in Figure 4, this time excluding the benchmarks with the highest and lowest speedups. We still see good speedups of 2.7x with 20,000 gates, and 3.1x with 25,000 gates.

A second observation we might make from the data in Table 1 and Figure 2 is that most speedup is obtained by moving the first few critical loops to FPGA - moving additional loops yields little additional speedup improvement. To see this trend more clearly, we continued the plot of Figure 2 for the top ten loops of each benchmark, assuming the remaining loops could be ideally sped up, meaning they could be implemented to execute in hardware in zero time. The results are shown in Figure 5. The vertical axis shows the cumulative ideal speedup as loops are moved to hardware. Shadowed points represent actual data, and non-shadowed points are ideal data. We can indeed see a leveling off effect. The biggest jump in speedup occurs within the first few loops. After that, subsequent loops tend to increase the speedup at a slower rate. Keep in mind that actual speedups for the latter loops would be *even less* – the figure shows ideal speedups for those latter loops. The implications of this observation are good for hardware/software partitioning - by moving just the most critical loops, we gain most of the possible speedup.

Table 2 lists the ideal speedups that would be obtained if every critical loop could be implemented in hardware in zero time. We see that even in the completely idea situation, the first few loops give most of the speedup.

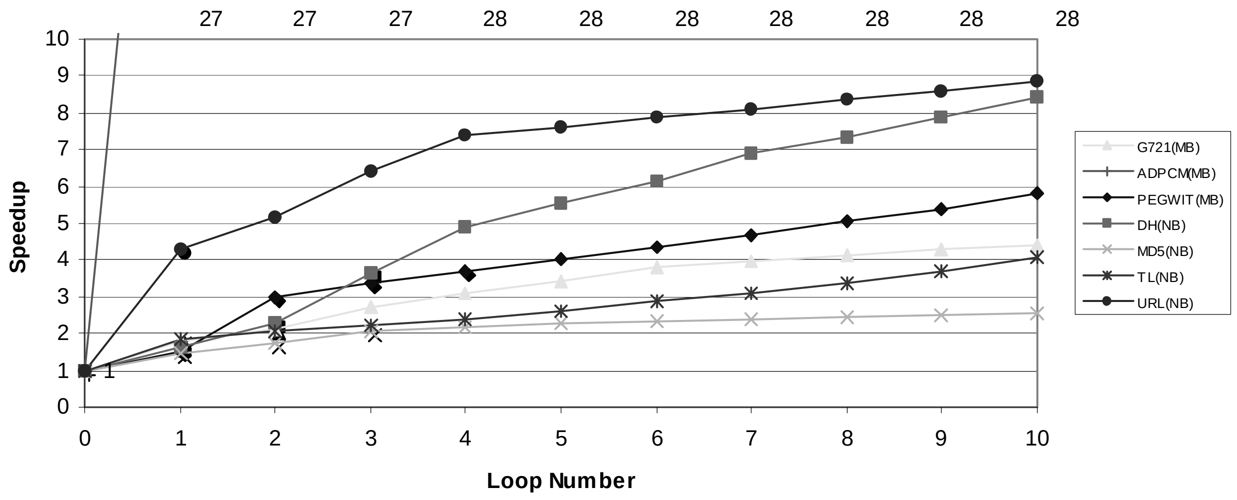
4. Conclusion

Partitioning critical software loops onto an on-chip FPGA yields impressive software speedups of 6.3x using a modestly sized FPGA of about 25,000 gates in the Mediabench and Netbench examples we tested. Most of the readily available speedup can be achieved within this 25,000 gate threshold. The implication of the speedup data for platform designers is that including even a modest amount of FPGA can yield good software improvements. As feature sizes continue to scale down, adding a 40,000 gate equivalent FPGA onto a microprocessor chip may become less and less significant. Furthermore, the implication for embedded system designers is that performing a straightforward hardware/software partitioning may be well worth the effort.

Table 2: Completely ideal speedup as loops are moved to hardware.

Benchmark	Ideal Cumulative Speedup with Subsequent Loops									
	1	2	3	4	5	6	7	8	9	10
G721	1.80	2.44	3.24	3.76	4.26	4.82	5.12	5.38	5.66	5.87
ADPCM	854.40	1229.39	1388.79	1594.01	1699.84	1806.98	1912.74	1994.41	2067.18	2127.34
Pegwit	1.55	3.42	3.94	4.42	4.97	5.43	5.99	6.58	7.18	7.88
DH	1.68	2.40	4.03	5.58	6.41	7.25	8.32	9.00	9.80	10.69
MD5	1.48	1.84	2.31	2.44	2.57	2.67	2.76	2.85	2.90	2.95
TL	2.04	2.24	2.46	2.68	2.95	3.26	3.58	3.96	4.41	4.97
URL	5.00	6.25	8.18	9.91	10.32	10.76	11.23	11.72	12.21	12.74

Figure 5: Actual speedups (shadowed) followed by ideal speedups as loops are moved to hardware.



References

[1] Altera, http://www.altera.com.

[2] Atmel, http://www.atmel.com.

[3] D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Repor #1342. June, 1997.

[4] J. Hauser, J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. IEEE Symposium on FPGAs Valley, CA, April 1997.

[5] J. Henkel. A low power hardware/software partitioning approach for core-based embedded systems. Proceedings of the 36th ACM/IEEE conference on Design automation conference, pp. 122 – 127,1999.

[6] C. Lee, M. Potkonjak and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, MICRO 1997.

[7] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. IEEE/ACM Design Automation Conference, pp 507-512, 2000.

[8] G. Memik, B. Mangione-Smith, W. Hu. Netbench: A Benchmarking Suite for Network Processors. CARES Technical Report 2001_2_01.

[9] W. Najjar, B. Draper, A.P.W. Böhm, R. Beveridge. The Cameron Project: High-Level Programming of Image Processing Applications on Reconfigurable Computing Machines. In PACT'98 - Workshop on Reconfigurable Computing. Paris, October 1998.

[10] K. A. Olukotun, R. Helaihel, J. Levitt, R. Ramirez. A software-hardware cosynthesis approach to digital system simulation. IEEE Micro, pp. 48-58, August 1994.

[11] Proceler, http://www.proceler.com.

[12] G. Stitt, B. Grattan, J. Villarreal, F. Vahid. Using On-Chip Configurable Logic to Reduce Embedded System Software Energy. IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, April 2002.

[13] Triscend, http://www.triscend.com.

[14] J. Villarreal, R. Lysecky, S. Cotterell, and F. Vahid. Loop Analysis of Embedded Applications. UC Riverside Technical Report UCR-CSE-01-03, 2001.

[15] M. Wirthlin, B. Hutchings. A dynamic instruction set computer. Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1995, pg. 99-107.

[16] Xilinx, http://www.xilinx.com.